(waitpid) and More On Signals

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

pollev.com/tqm

- Any questions regarding penn-parser or penn-shredder

Administrivia

- Project 0 penn-parser:
  - Autograder is up, due Friday @ 11:59pm
  - Actual due date: submit with penn-shredder, but you need to finish it before penn-shredder will work anyways.
  - Your first C programming assignment

- Project 1 penn-shredder:
  - Due Friday Feb 02nd
  - You need penn-parser to complete it
  - Is not much more once you have implemented penn-parser
  - Should have everything you need, this lecture may help

Administrivia

- There will be a check-in due next week
  - Due before Tues @ 5pm
- First “recitation” was yesterday, there will be another again next week
  - Tentatively Monday next week, waiting on room reservation
  - Covers topics that should help with projects, and then have open OH afterwards.
- Pre-semester survey:
  - “due” wed Jan 31 TOMORROW
  - Just a short survey
  - Please do it

Lecture Outline

- wait & waitpid & busy waiting
- Signals refresher
- Sigset
- Signal blocking vs signal ignoring
- Signal Safety
- Sigsuspend
- Process diagram updated

wait() status

- **`status`** output from **`wait()`** can be passed to a macro to see what changed
- **`WIFEXITED()`** true iff the child exited nomrally
- **`WIFSIGNALED()`** true iff the child was signaled to exit
- **`WIFSTOPPED()`** true iff the child stopped

- See example in `status_check.c`

More: `waitpid()`

```
pid_t waitpid(pid_t pid, int *wstatus,
              int options);
```

- Calling process waits for a child process (specified by **`pid`**) to exit
  - Also cleans up the child process
- Gets the exit status of child process through output parameter **`wstatus`**
- **`options`** are optional, pass in `0` for default options in *most* cases
- Returns process ID of child who was waited for or `-1` on error

Non blocking wait w/ `waitpid()`

- ```
  pid_t waitpid(pid_t pid, int *wstatus,
                int options);
  ```
  - Can pass in WNOHANG for **`options`** to make **`waitpid()`** not block or "hang".
  - Returns process ID of child who was waited for or **`-1`** on error or 0 if there are no updates in children processes and WNOHANG was passed in
  ```

CPU Utilization

- When a process is in a blocked state, it will not be run by the scheduler and thus will not use the CPU

- When analyzing performance, one thing people care about is making maximal use of the CPU. The CPU is what is executing our instructions.
  - Avoiding wasting CPU cycles on things that don't matter
  - Make sure the CPU is running as much instructions (that matter) as possible

discuss

```
 8 int main() {
 9   int status = 0;
10   /* fork another process */
11   pid_t pid = fork();
12
13   if (pid < 0) {
14     /* error occurred */
15     fprintf(stderr, "Fork Failed");
16     return EXIT_FAILURE;
17   } else if (pid == 0) {
18     /* child process */
19     char* args[] = {"/bin/sleep", "10", NULL};
20     execvp(args[0], args);
21     return EXIT_FAILURE;
22   }  else {
23     pid_t res = waitpid(pid, &status, 0);
24     while (res == 0) {
25       // no status update yes
26       printf("waiting...\n");
27       res = waitpid(pid, &status, WNOHANG);
28     }
29     printf("Done!\n");
30     return EXIT_SUCCESS;
31   }
32 }
```

What does this code print?

Poll Everywhere

pollev.com/tqm

```
 8 int main() {
 9   int status = 0;
10   /* fork another process */
11   pid_t pid = fork();
12
13   if (pid < 0) {
14     /* error occurred */
15     fprintf(stderr, "Fork Failed");
16     return EXIT_FAILURE;
17   } else if (pid == 0) {
18     /* child process */
19     char* args[] = {"/bin/sleep", "10", NULL};
20     execvp(args[0], args);
21     return EXIT_FAILURE;
22   }  else {
23     pid_t res = waitpid(pid, &status, 0);
24     while (res == 0) {
25       // no status update yes
26       printf("waiting...\n");
27       res = waitpid(pid, &status, WNOHANG);
28     }
29     printf("Done!\n");
30     return EXIT_SUCCESS;
31   }
32 }
```

If we change line 23 to use WNOHANG, what is printed? Which code is likely better?

```
pid_t res = waitpid(pid, &status, WNOHANG
```

Blocking

- Do we always want to block?
  - In the simple cases, yes
  - In more complex cases (like in penn-shell), it may not be desirable

- If we don't block, that means we can make progress on other tasks. If we had blocked, those other tasks are also waiting on that task
  - More on this later in the semester when we talk about threads
  - This idea is related to asynchronous programming

Busy Waiting

- **Busy Waiting**: when code repeatedly checks some condition, waiting for the condition to be satisfied
  - Sometimes called *Spinning*, like the phrase "spinning your wheels"

- We just did this before, see `no_hang.c`

- Demo: running `no_hang` and using the terminal command `top` to see the CPU utilization

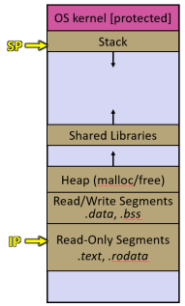
Lecture Outline

- wait & waitpid & busy waiting
- **Signals refresher**
- Sigset
- Signal blocking vs signal ignoring
- Signal Safety
- Sigsuspend
- Process diagram updated

Diagram: signals

User Processes

./example
pid = 100



OS

Process Table

100

PCB: example

```
id = 100
status = blocked
sig_dispositions = {
  SIGTOU: SIG_DFL,
  SIGALRM: SIG_IGN,
  SIGINT: handler()
}
```

Diagram: signals

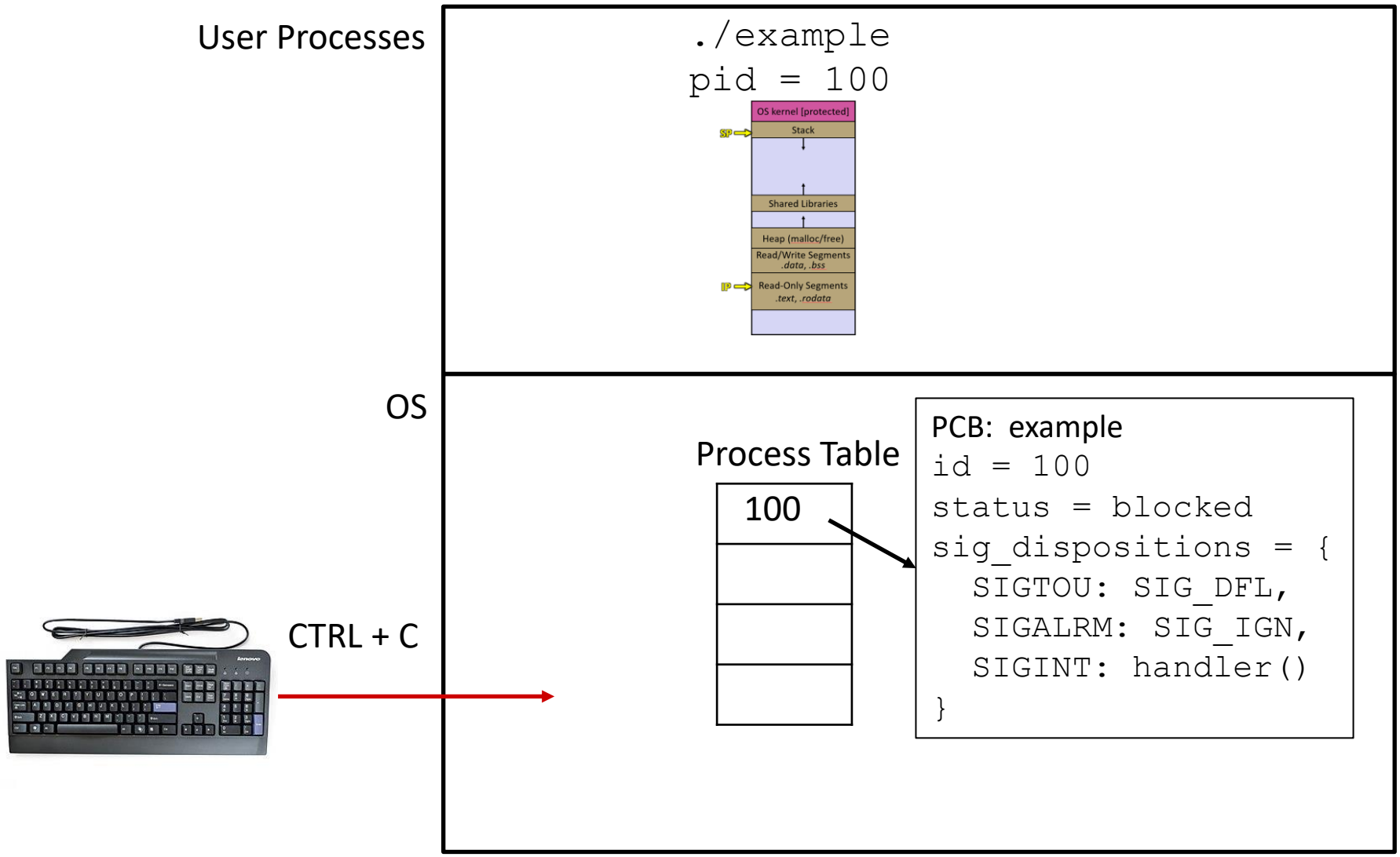
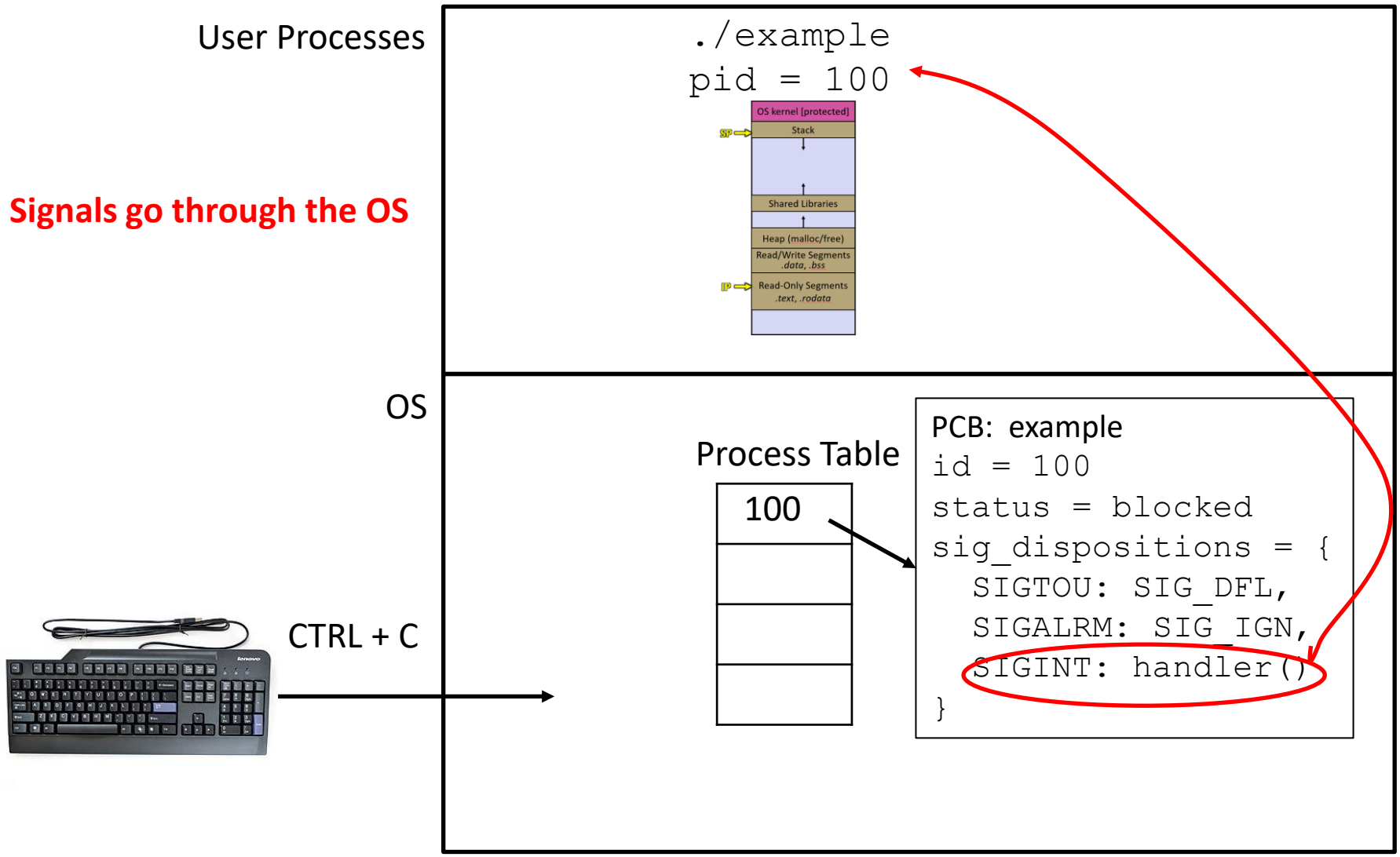


Diagram: signals



kill()

- Can send specific signals to a specific process manually
- ```
  int kill(pid_t pid, int sig);
  ```
- pid: specifies the process
- sig: specifies the signal

- Example: `kill(child, SIGKILL);`

- Put this at the top of your penn-shredder.c file (before #includes) to use **kill()**

```
#define _POSIX_C_SOURCE 1
```
```

Diagram: signals between processes

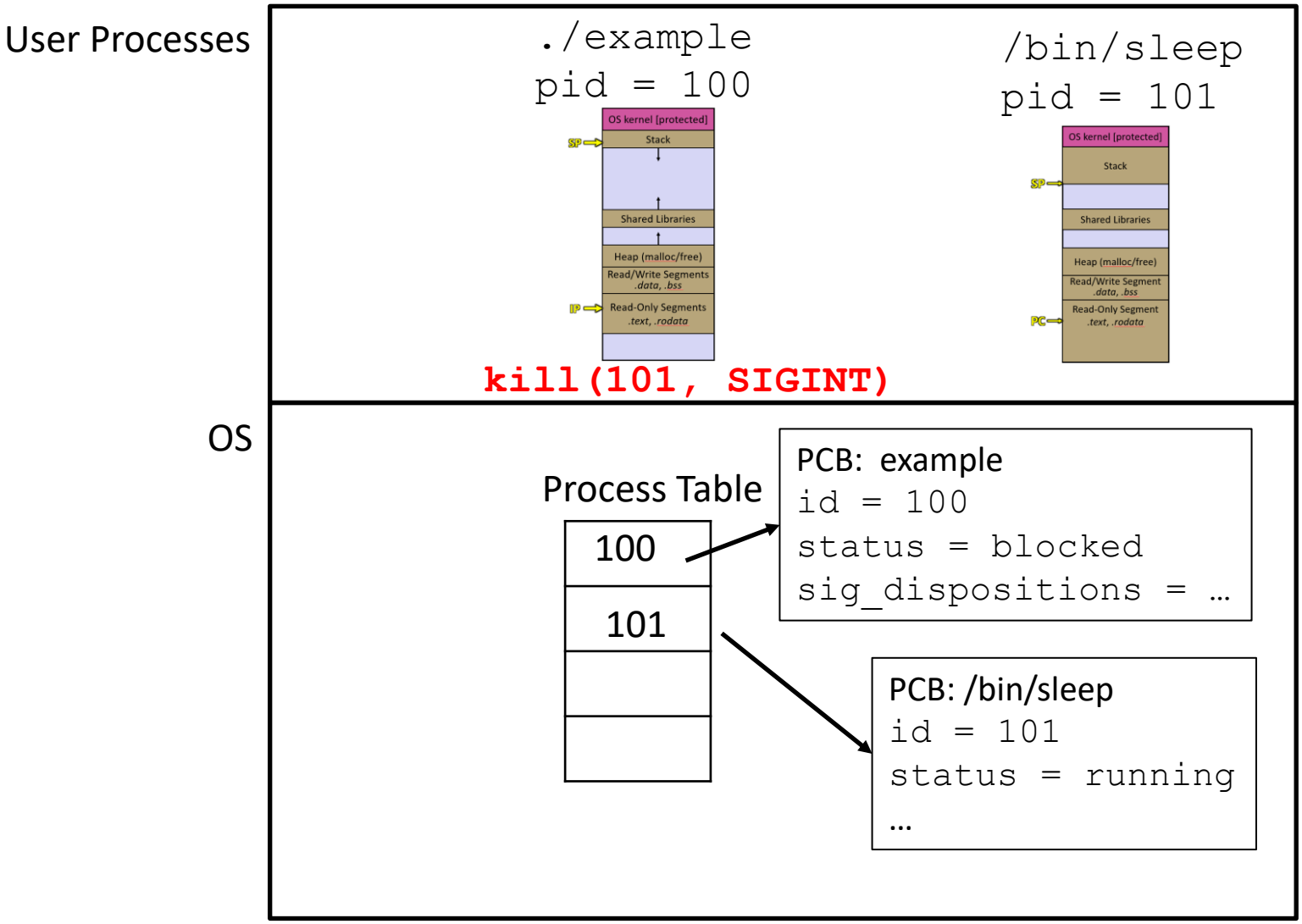
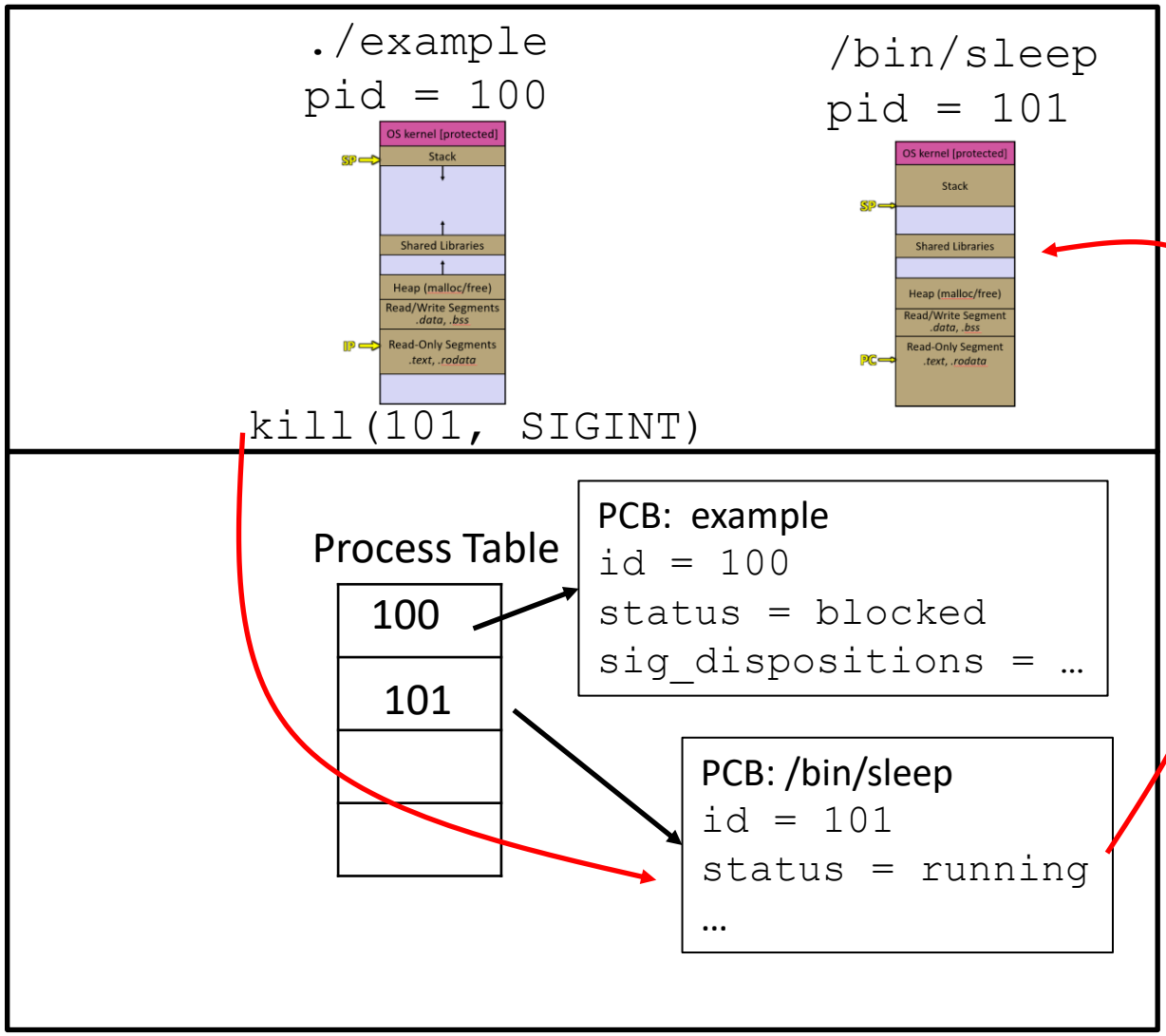


Diagram: signals between processes



When one process tries to send a signal to another, it goes through the OS

Good rule of thumb: If a process wants to interact with another process, it does so through the OS.

The OS tries to make sure processes stay "safe" in their interactions

Demo impatient.c

- See `impatient.c`
  - Parent forks a child, checks if it finishes every second for 5 seconds, if child doesn't finish send SIGKILL

  - LOOKS SIMILAR TO WHAT YOU ARE DIONG IN penn-shredder. DO NOT COPY THIS
    - **`waitpid()`** IS NOT ALLOWED
    - USING **`sleep()`** AND **`alarm()`** TOGETHER CAN CAUSE ISSUES

Signals can interrupt other signals

- See code demo: `interrupt.c`
  - Handler registered for SIGALRM and SIGINT
  - Once SIGALRM goes off, it continuously loops and prints
  - SIGINT can be input and run its handler even if SIGALRM was running its handler

SIGCHLD handler

- Whenever a child process updates, a **`SIGCHLD`** signal is received, and by default ignored.

- You can write a signal handler for **`SIGCHLD`**, and use that to help handle children update statuses: allowing the parent process to do other things instead of calling **`wait()`** or **`waitpid()`**

- Relevant for proj2: **`penn-shell`**

Lecture Outline

- wait & waitpid & busy waiting
- Signals refresher
- Sigset
- **Signal blocking vs signal ignoring**
- Signal Safety
- Sigsuspend
- Process diagram updated

Previously: Execution Blocking

- When a process calls **`wait()`** and there is a process to wait on, the calling process **blocks**
- If a process **blocks** or is **blocking** it is not scheduled for execution.
  - It is not run until some condition "unblocks" it
  - For **`wait()`**, it unblocks once there is a status update in a child
- This happens frequently when a system call is made, that calling process will block till the system call can be completed.
- **This is DIFFERENT than signal blocking**

Signal Blocking

- A process has some set of signals called a "signal mask"
  - Signals in that set/mask are "**blocked**"
  - Blocked signals mean that the signal is temporarily paused from being delivered, **instead that signal is "delayed" until the process eventually unblocks that signal**

- Common mistake: thinking this is the same as calling `signal(SIG____, SIG_IGN);`
  This function call marks the signal as ignored, which means a signal delivered during this time is completely ignored, never delivered later.

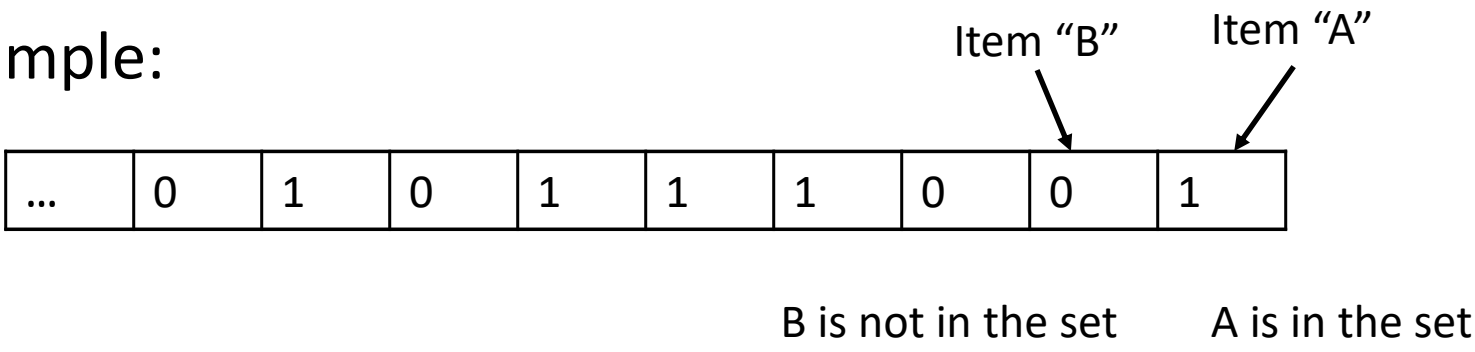
- **REMINDER: Different from a process "blocking"**

Aside: a way to implement a set in C

- If we have a fixed number of items that can possibly be in the set, then we can use a **bitset**

- Have at least N bits, each item corresponding to a single bit.
  - Each items assigned bit can either be a 0 or a 1, 0 to indicate absence in the set, 1 to indicate presence in the set

- Example:



pollev.com/tqm

- If we have 39 signals, how many bits do we need to have a bitset to represent all signals? How many bytes?

sigset_t

- `int sigemptyset(sigset_t* set);`
  - Initializes a sigset_t to be empty
- `int sigaddset(sigset_t* set, int signum);`
  - Adds a signal to the specified signal set
- More functions & details in man pages
  - (`man sigemptyset`)
- Example snippet:

```
sigset_t mask;
if (sigemptyset(&mask) == -1) {
  // error
}
if (sigaddset(&mask, SIGINT) == -1) {
  // error
}
```

sigprocmask()

- ```
  int sigprocmask(int how, const sigset_t* set,
                  sigset_t* oldset);
  ```
  - Sets the process mask to be the specified process "block" mask
  - Three arguments, how do we use them?

pollev.com/tqm

- Look at the man page, how do we complete this code?
  - **man sigprocmask**

```
sigset_t mask;
if (sigemptyset(&mask) == -1) { // error }
if (sigaddset(&mask, SIGINT) == -1) { // error }

// how do we block SIGINT?
```
```

Demo: `delay_sigint.c`

- Demo: `delay_sigint.c`
  - blocks the signal SIGINT so that if CTRL + C is typed in the first 5 seconds, it doesn't get processed till after the first 5 seconds of the program running
  - CTRL + C after the first 5 seconds works as normal

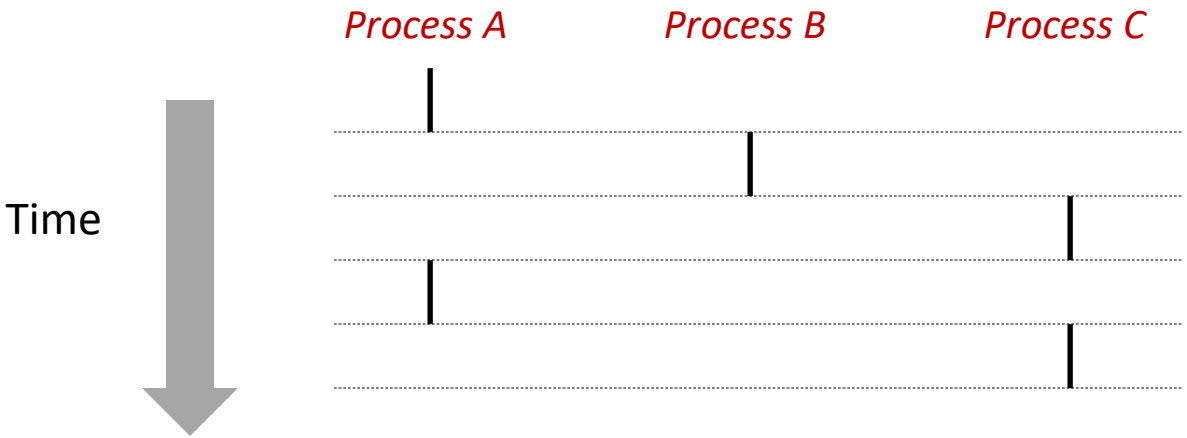
Lecture Outline

- wait & waitpid & busy waiting
- Signals refresher
- Sigset
- Signal blocking vs signal ignoring
- **Signal Safety**
- Sigsuspend
- Process diagram updated

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on **single core**):
  - Concurrent: A & B, A & C
  - Sequential: B & C

Note how at any specific moment in time only one process is running

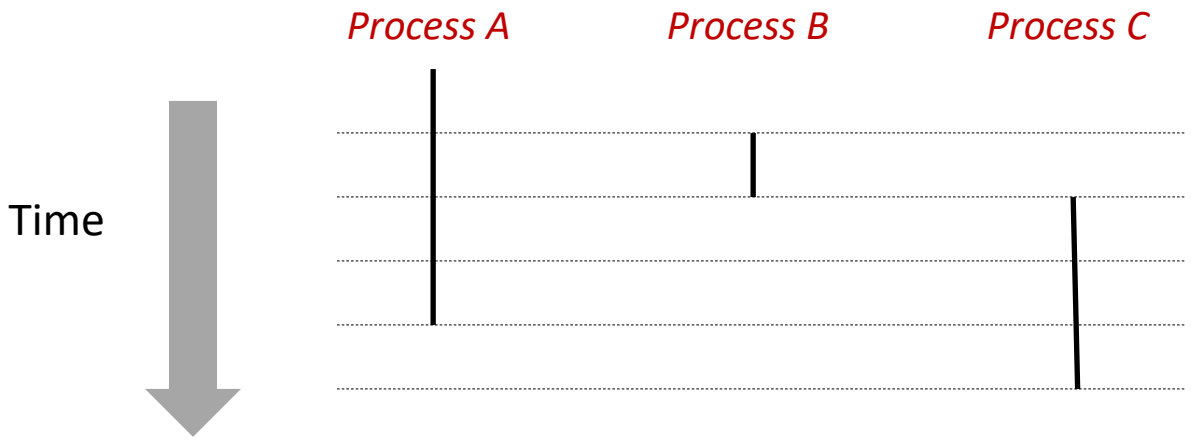


Black line indicates that the process is running during that time

Parallel Processes

Assuming more than one CPU/CORE

- Each process is a logical control flow.
- Two processes run parallel if their flows overlap at a specific point in time. (Multiple instructions are performed on the CPU at the same time
- Examples (running on dual core):
  - Parallel: A & B, A & C
  - Sequential: B & C



Critical Sections

- There can be issues when one or more resources are accessed concurrently that causes the program to be put in an unexpected, invalid, or error state.

  These sections of code where these accesses happen, called ***critical sections***, need to be protected from concurrent accesses happening during it

- With concurrent processes accessing OS resources, the OS will handle critical sections for us
- **Even if we have one process**, we can have signal handlers execute at any time, leading to possible concurrent access of memory, which is not default protected for us

Poll Everywhere

pollev.com/tqm

```
// assume this works
void list_push(list* this, float to_push) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) exit(EXIT_FAILURE);
  node->value = to_push;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
void handler(int signo) {
  list_push(list, 4.48);
}

int main(int argc, char* argv[]) {
  signal(SIGINT, handler);
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
}
```

This code is broken. It compiles, but it doesn't ***always*** do what we want. Why?

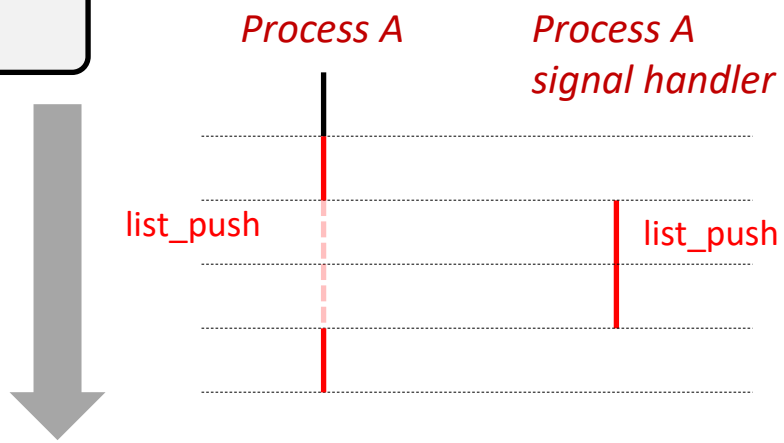
- Assume we have implemented a linked list, and it works
- Assume **`list`** is an initialized global linked list

Critical Section

- This code is problematic since there is a critical section

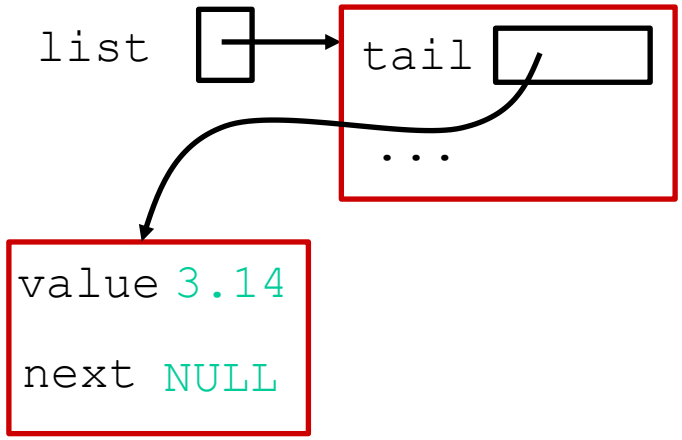
```
void handler(int signo) {
  list_push(list, 4.48);
}

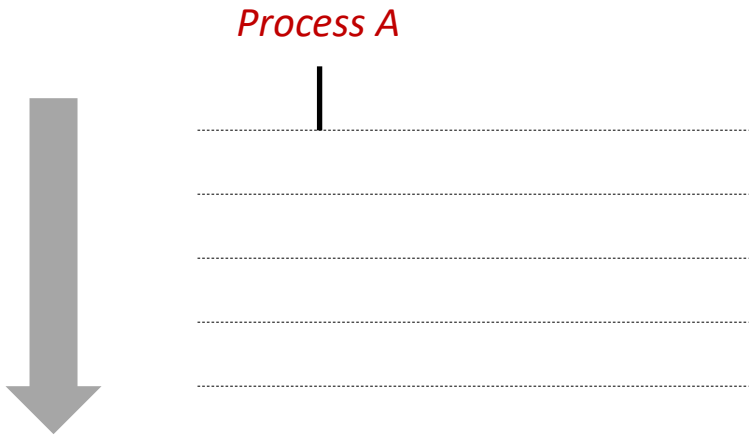
int main(int argc, char* argv[]) {
  signal(SIGINT, handler);
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
}
```



Critical Section Walkthrough

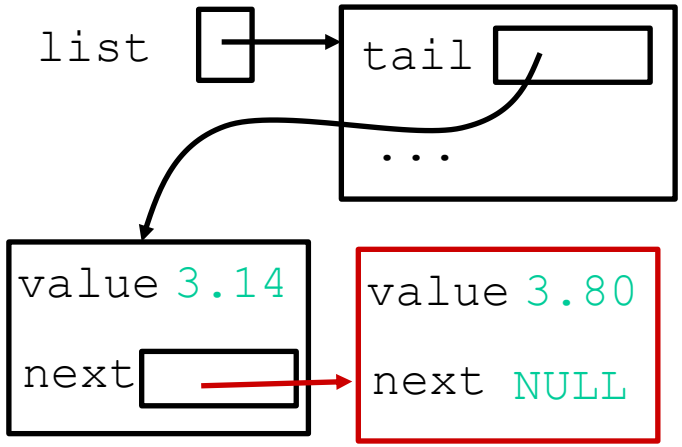
```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

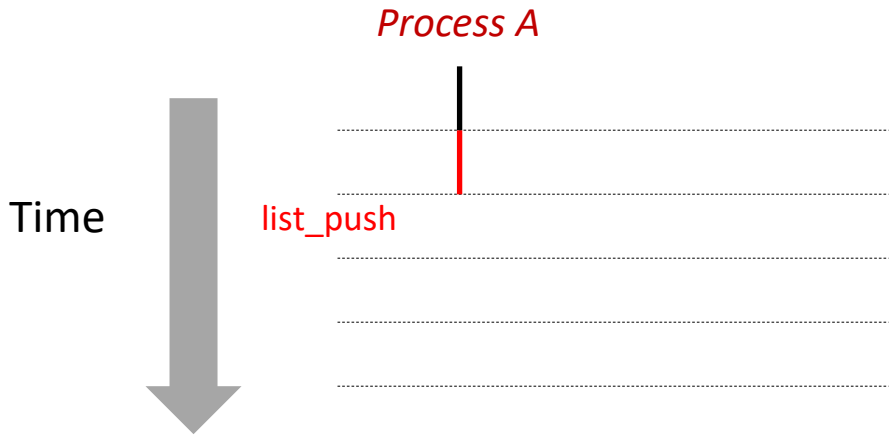




Critical Section Walkthrough

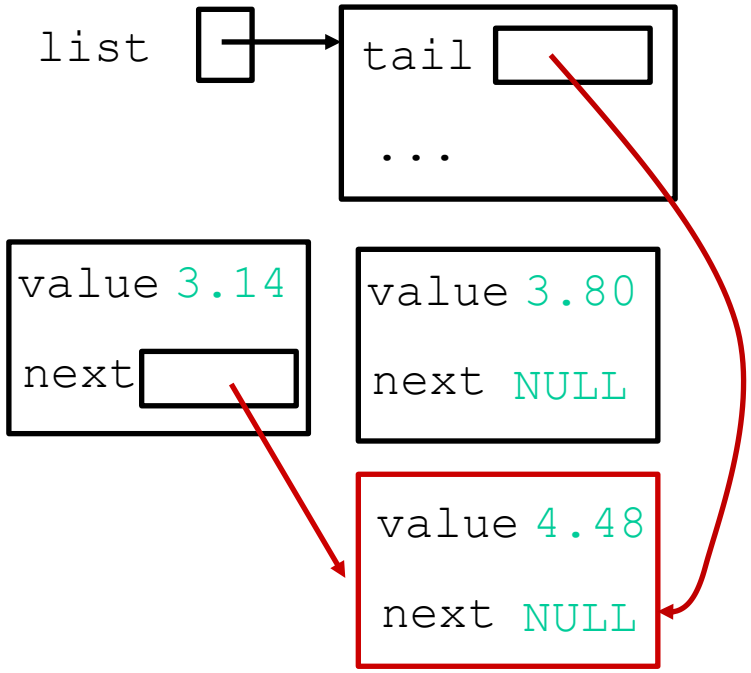
```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```



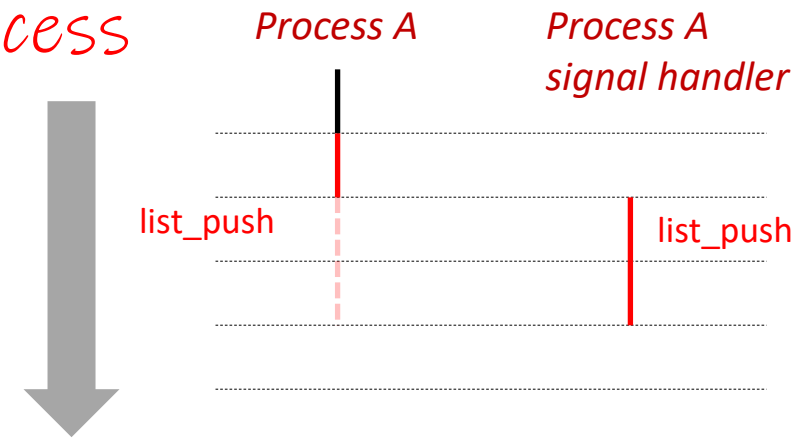


Critical Section Walkthrough

```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

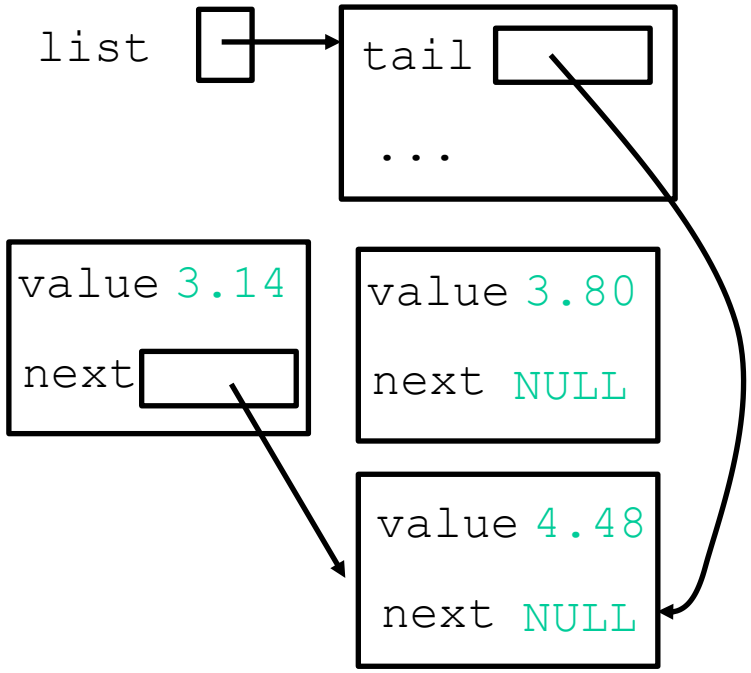


Signal handler interrupts and runs list_push while the process is normally running list_push

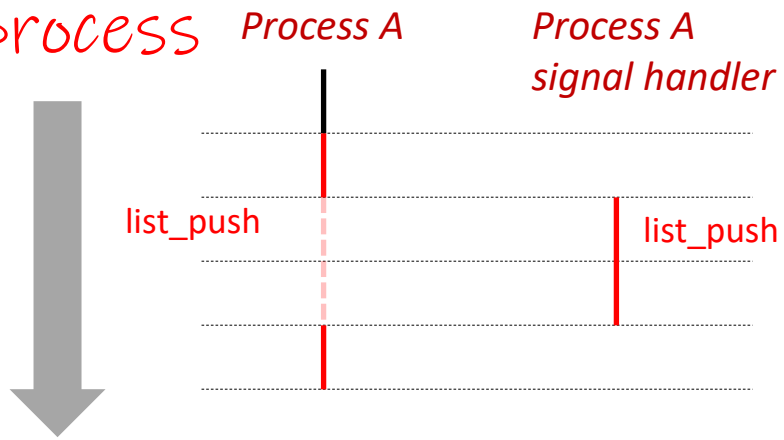


Critical Section Walkthrough

```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

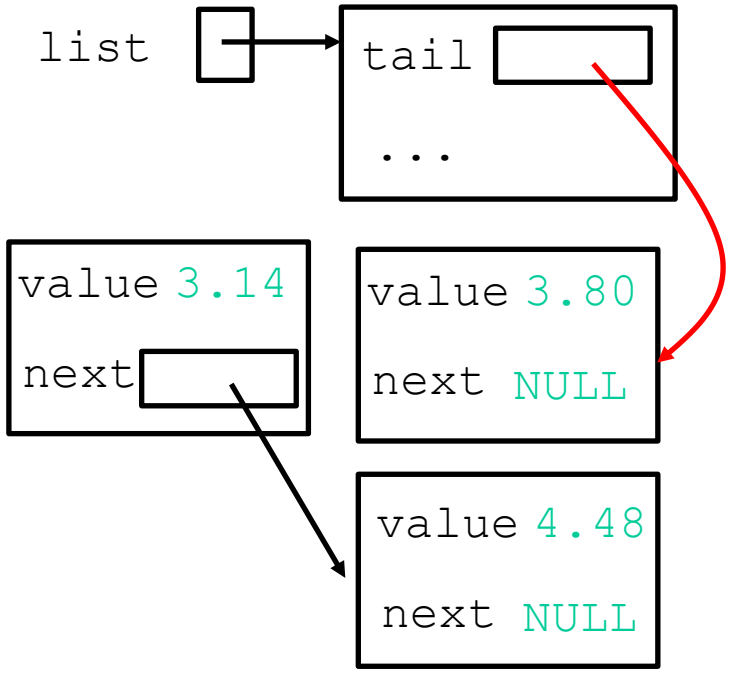


Signal handler finishes, and we return to running the main process normally...



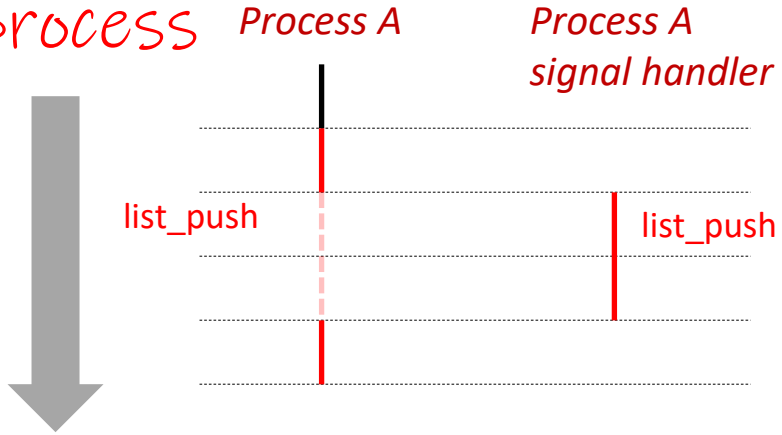
Critical Section Walkthrough

```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```



Signal handler finishes, and we return to running the main process normally

and we end up in an invalid linked list state...



Poll Everywhere

pollev.com/tqm

```
// assume this works
void list_push(list* this, float to_push) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) exit(EXIT_FAILURE);
  node->value = to_push;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
void handler(int signo) {
  list_push(list, 4.48);
}

int main(int argc, char* argv[]) {
  signal(SIGINT, handler);
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
}
```

- How can we fix this code?

Signal Safety

- From **`man 7 signal-safety`**
  - To avoid problems with unsafe functions, there are two possible choices:
    - (a) Ensure that (1) the signal handler calls only async-signal- safe functions, and (2) the signal handler itself is reentrant with respect to global variables in the main program.
      - Prefer this when possible
    - (b) Block signal delivery in the main program when calling functions that are unsafe or operating on global data that is also accessed by the signal handler.
      - Notably: printf, malloc, free, and many functions are not signal safe
      - We can do this with sigprocmask, but (a) is preferred when possible
  - Read more by typing `**`man 7 signal-safety`** into the terminal or google

Lecture Outline

- wait & waitpid & busy waiting
- Signals refresher
- Sigset
- Signal blocking vs signal ignoring
- Signal Safety
- **Sigsuspend**
- Process diagram updated

sigsuspend()

- Instead of busy waiting and wasting CPU cycles (that c an be used by other processes), we can block/suspend process execution instead

- ```
  int sigsuspend(const sigset_t* mask);
  ```
  - Temporarily replaces process mask with specified one and suspends execution till a signal that is not blocked is delivered.
  - If signal is caught by a handler, then after handler code will return from sigsuspend and the process signal mask will be restored

- Demo: `suspend_sigint.c`
  - Compare to previous code: `delay_sigint.c`
  - Less CPU resources used ☺
  ```

volatile sig_atomic_t

- If you need to communicate with a signal handler, we have been using global variables...
  - Modifying global variables is generally unsafe in signals.

- In “real world” code if you want to modify a signal handler, you should use global variable type: volatile sig_atomic_t
  - `voltaile sig_atomic_t` is an integer type

- We will not enforce this in these projects, but we felt like it was worth letting yo know

Lecture Outline

- wait & waitpid & busy waiting
- Signals refresher
- Sigset
- Signal blocking vs signal ignoring
- Signal Safety
- Sigsuspend
- **Process diagram updated**

Stopped Jobs

- Processes can be in a state slightly different than being blocked. *// This is relevant for* `penn-shell`
  - When a process gets the signal `SIGSTOP`, the process will not run on the CPU until it is resumed by the `SIGCONT` signal

- Demo:
  - In terminal: `ping google.com`
  - Hit `CTRL + Z` to stop
  - Command: "`jobs`" to see that it is still there, just stopped
  - Can type either "`%<job_num>`" or "`fg`" to resume it

Process State Lifetime