Signals & Critical Sections

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Poll Everywhere

pollev.com/tqm

- How aare u

Administrivia

- Proj0 (penn-shredder) Due **yesterday** @ 11:59 pm
  - **This assignment is done on your own**
  - **Late days still exist though (and they are applied automatically)**
- Peer Evaluation & Project1 to be released later this week
  - **Find a partner and sign up in a group on canvas**
  - Decent indicator of good partner for a pair: similar work ethic
- Project 1 Demo and Q&A in next lecture
- Check-in Quiz 2 Due in ~1 week

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

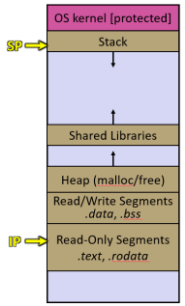
Lecture Outline

- **Signal high level view**
- Signal Blocking
  - `sigset_t` & `sigprocmask`
- Critical section & blocking
- Updated process state diagrams: stop & continue
- `sigsuspend` & busy waiting

Diagram: signals

User Processes

./example
pid = 100



OS

Process Table

100

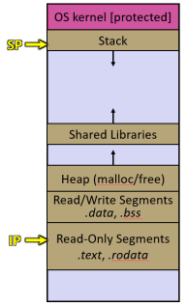
PCB: example

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

Diagram: signals

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()
}

CTRL + C

Diagram: signals

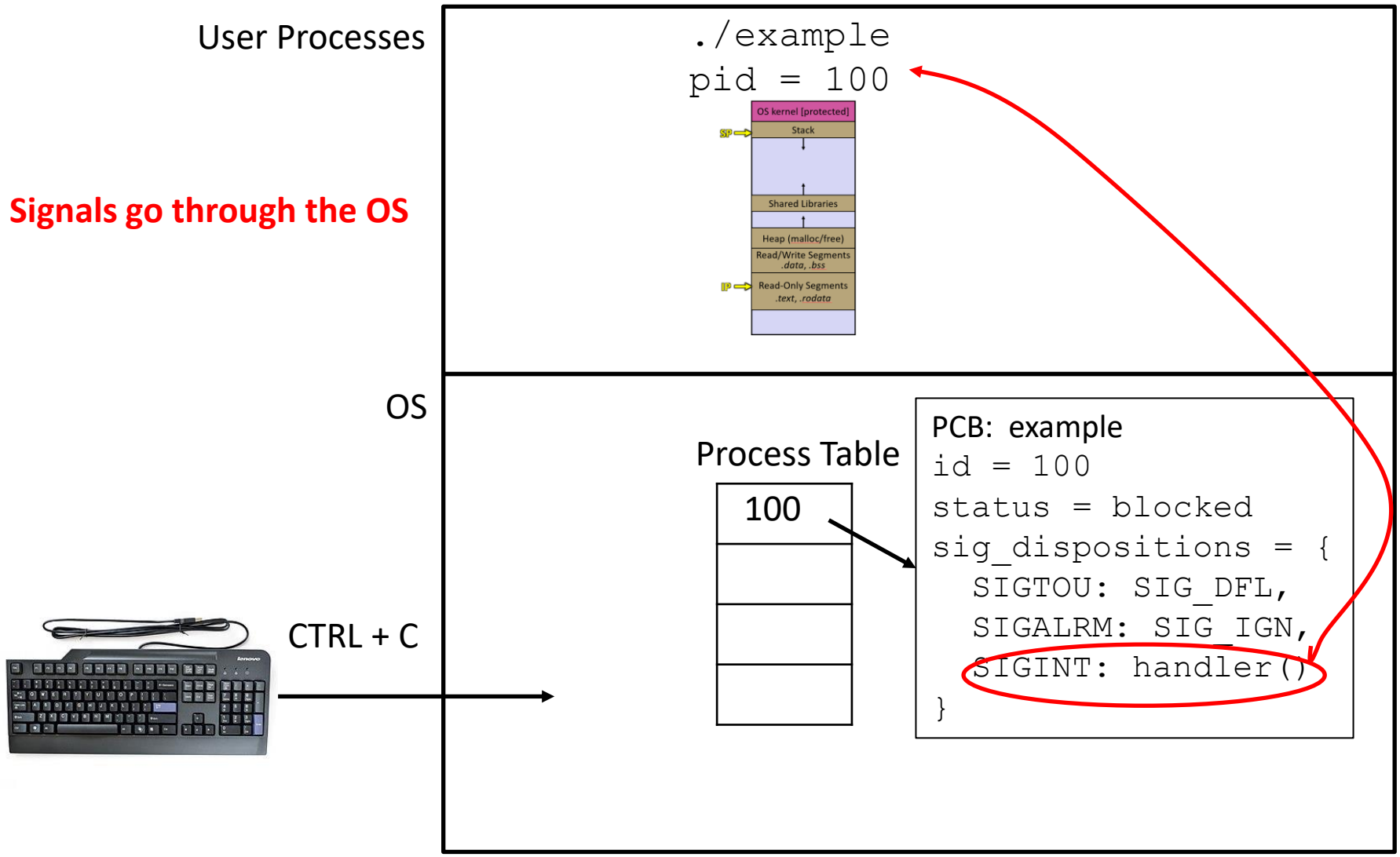


Diagram: signals between processes

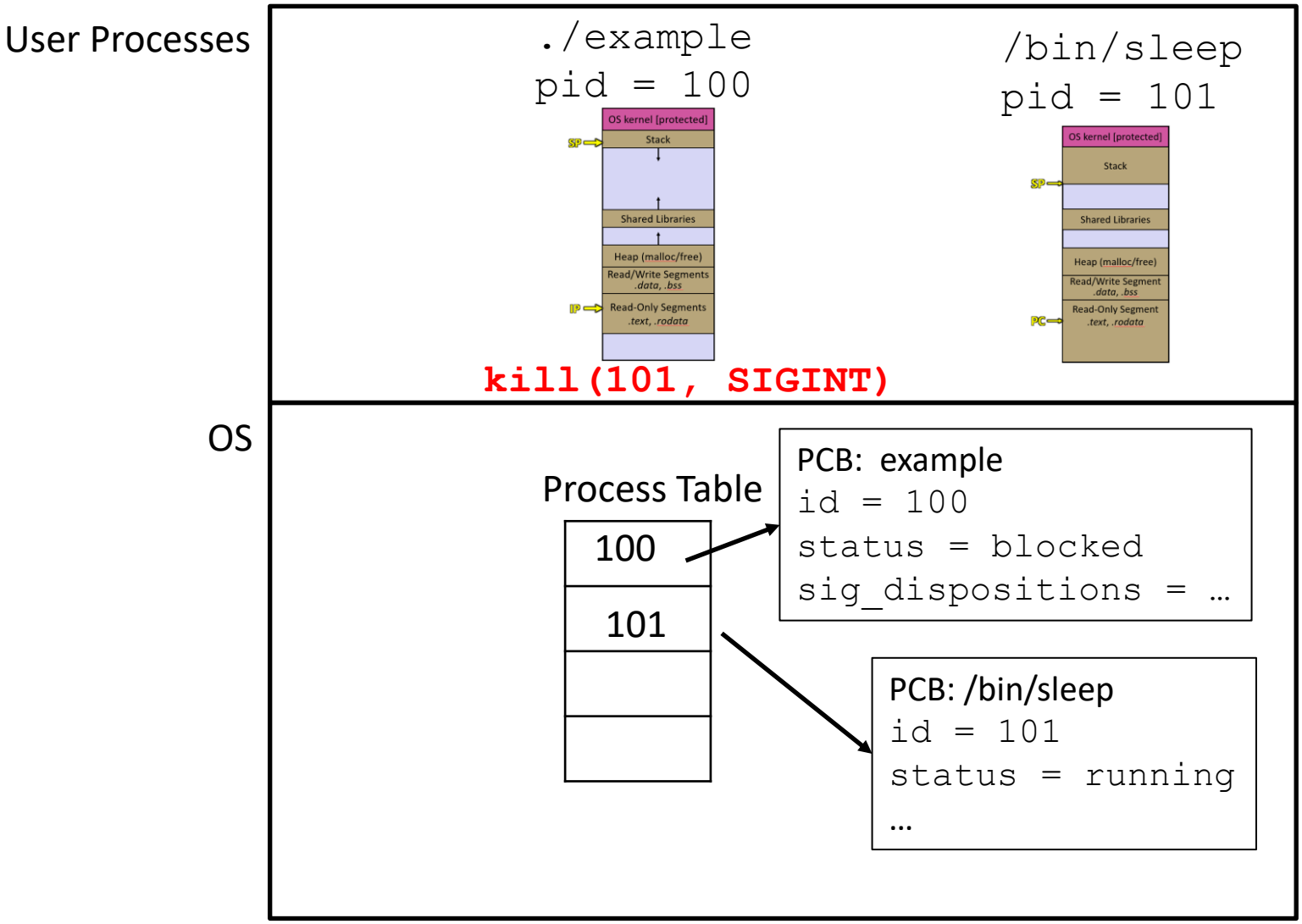
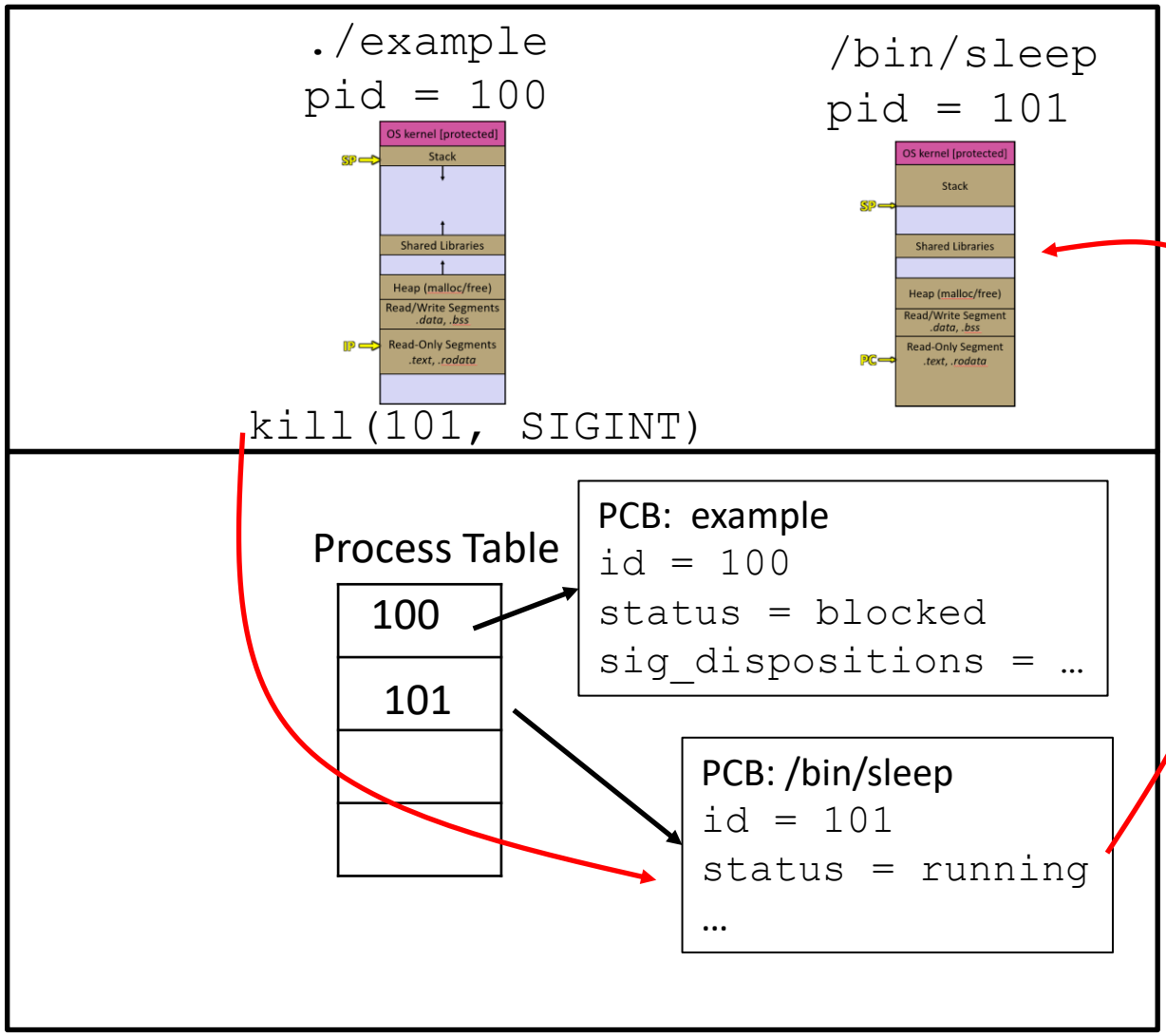


Diagram: signals between processes



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

Signals can interrupt other signals

- See code demo: `signal_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

Lecture Outline

- Signal high level view
- **Signal Blocking**
  - **`sigset_t` & `sigprocmask`**
- Critical section & blocking
- Updated process state diagrams: stop & continue
- `sigsuspend` & busy waiting

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.

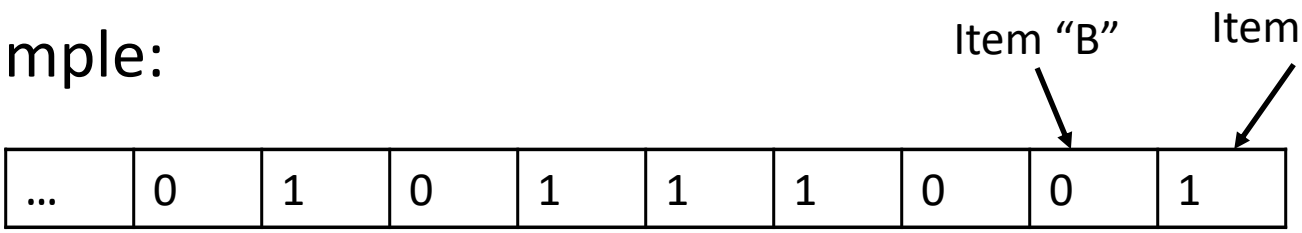
- **REMINDER: Different from a process "blocking"**

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

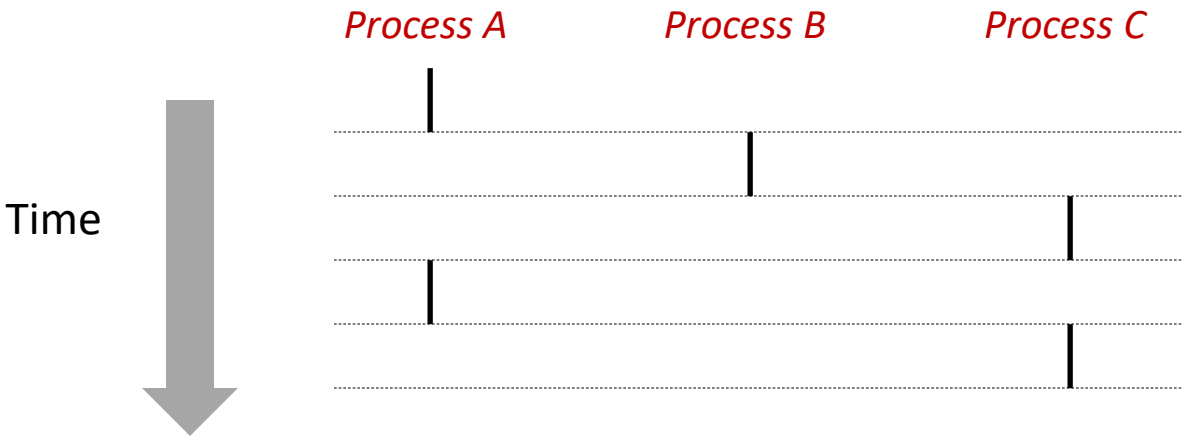
Lecture Outline

- Signal high level view
- Signal Blocking
  - `sigset_t` & `sigprocmask`
- **Critical section & blocking**
- Updated process state diagrams: stop & continue
- `sigsuspend` & busy waiting

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

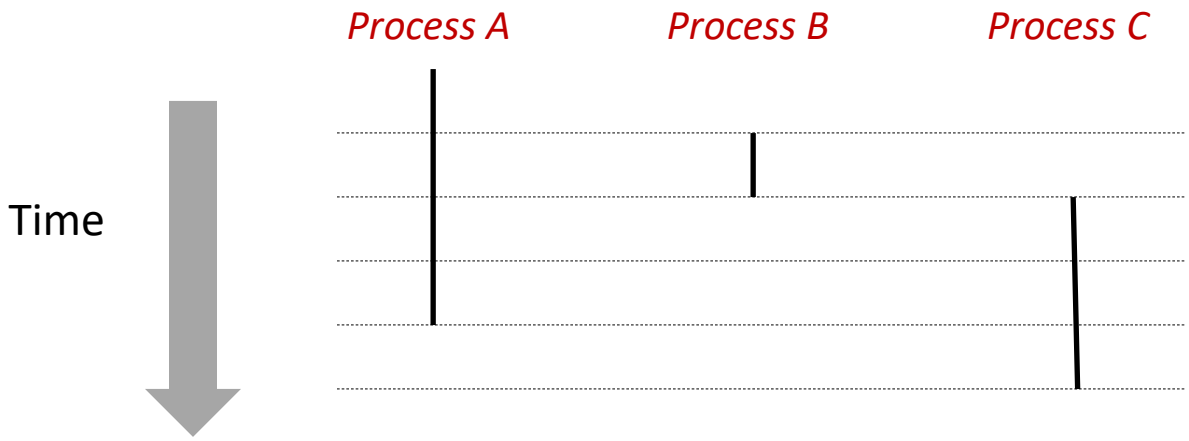


Black line indicates that the process is running during that time

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 a resources is accessed concurrently that causes the resource to be put in an invalid or error state.

  These sections of code, called **critical sections**, need to be protected from concurrent access 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

Remember this poll?

```
// 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, NaN);
}

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?

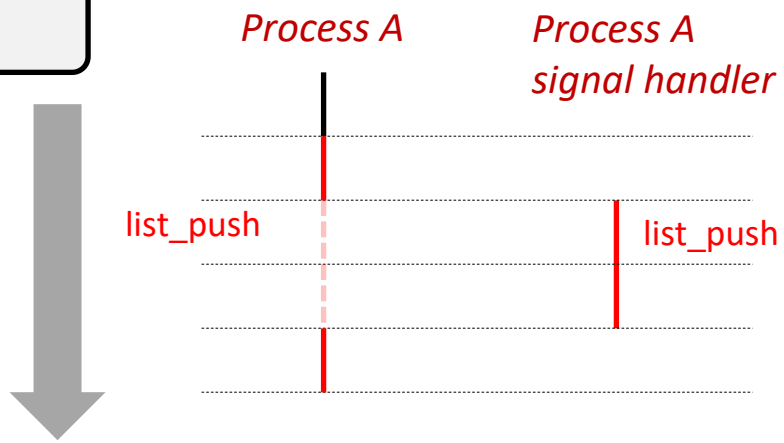
- Assume we have implemented a linked list, and it works
- Assume **`list`** is an initialized global linked list

Remember this poll?

- This code is problematic since there is a critical section

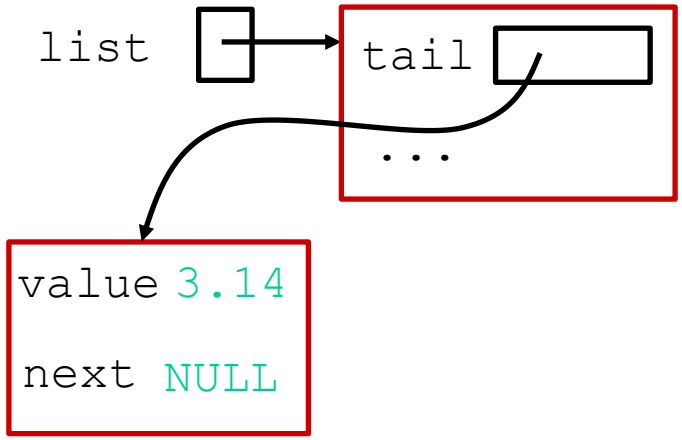
```
void handler(int signo) {
  list_push(list, NaN);
}

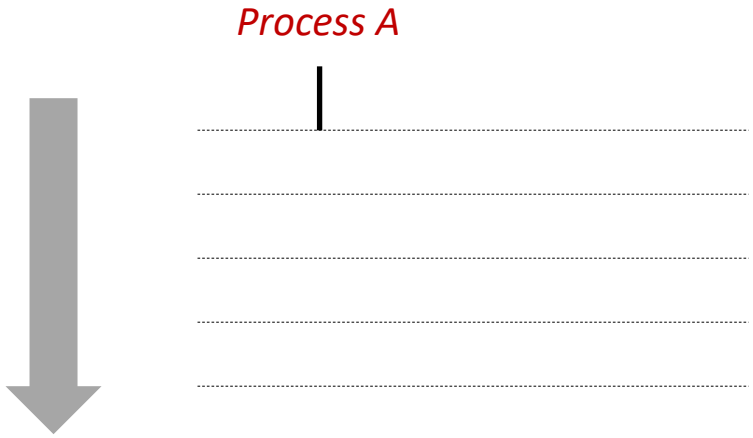
int main(int argc, char* argv[]) {
  signal(SIGINT, handler);
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
}
```



Critical Section Walkthrough

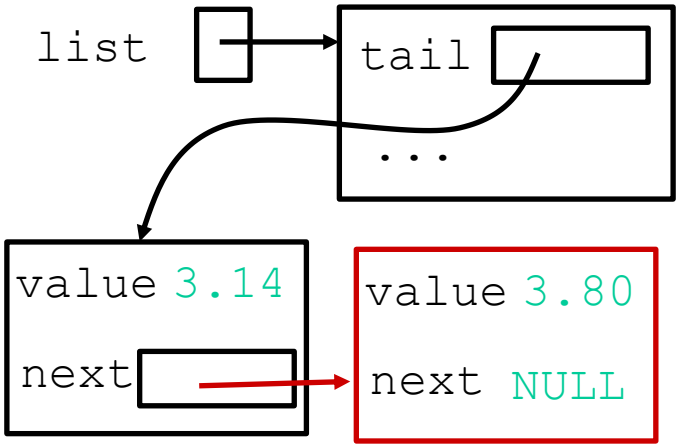
```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

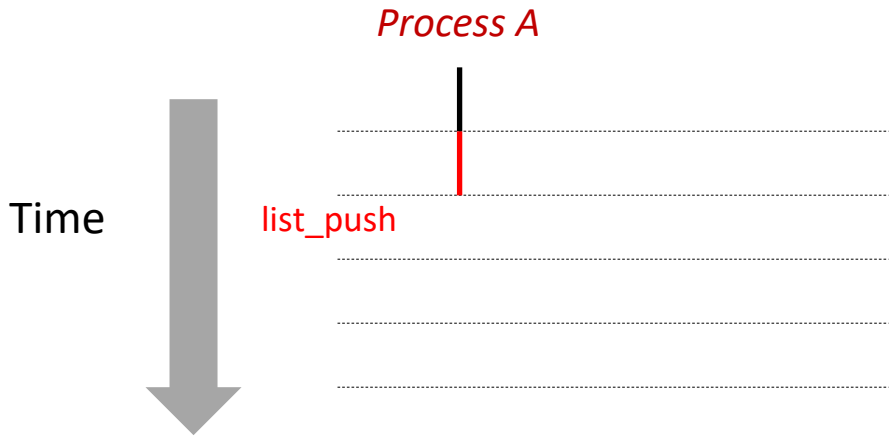




Critical Section Walkthrough

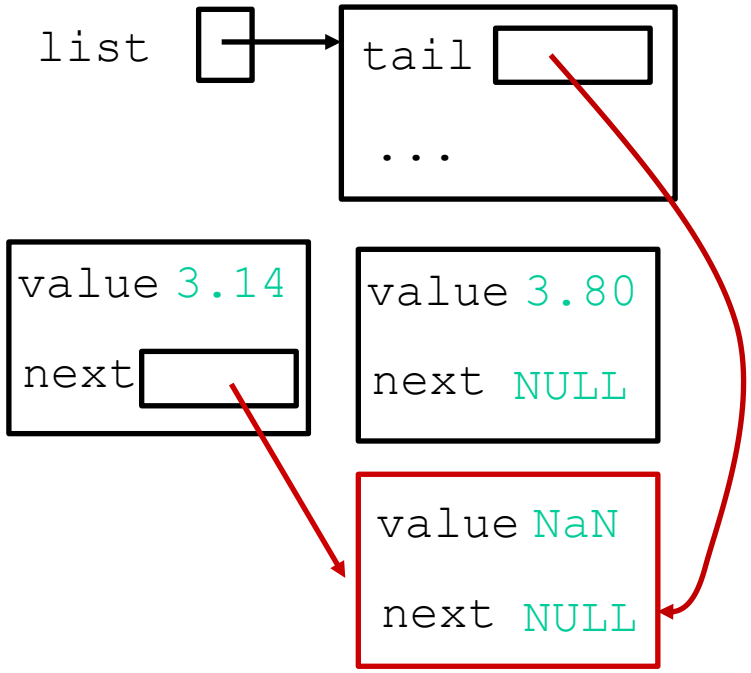
```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```



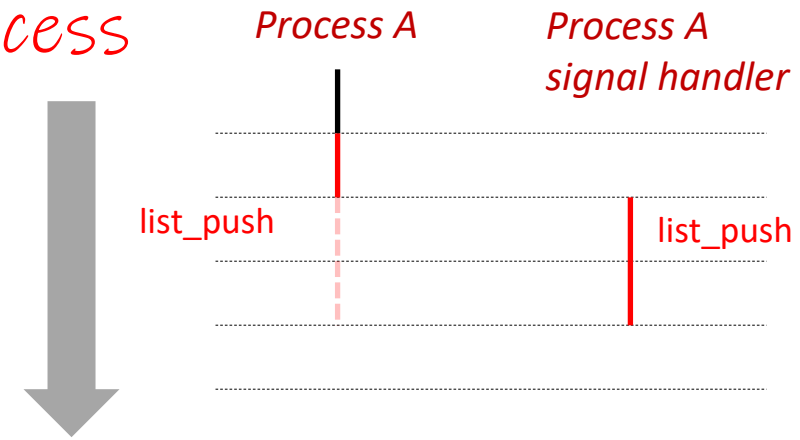


Critical Section Walkthrough

```
// 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;
}
```

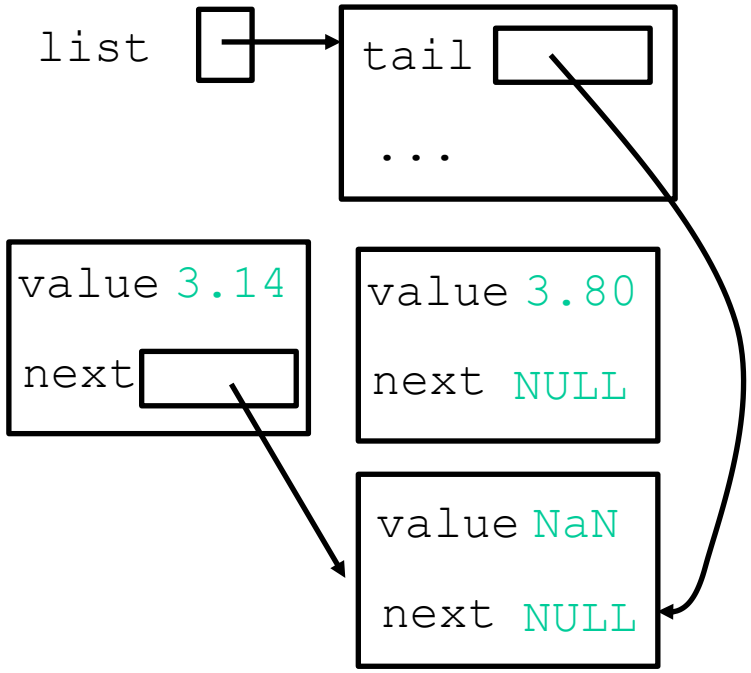


Signal handler interrupts and runs list_push while the process is normally running list_push

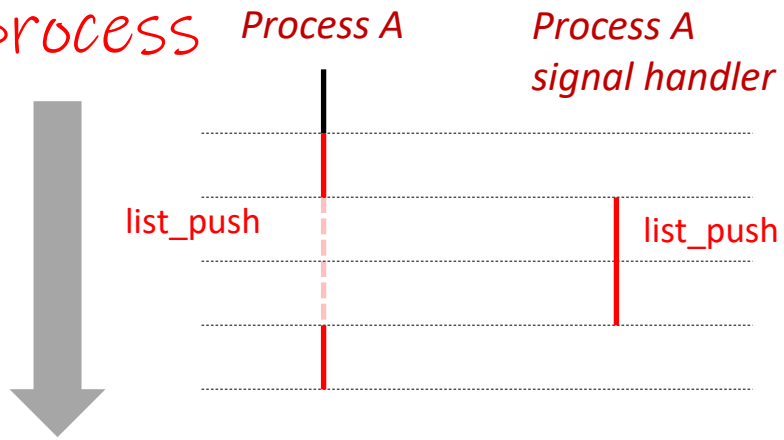


Critical Section Walkthrough

```
// 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;
}
```

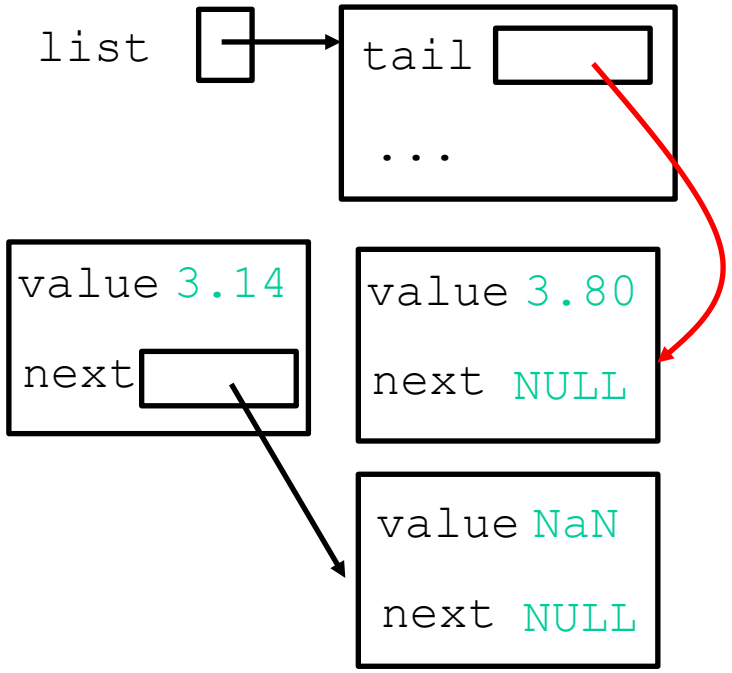


Signal handler finishes, and we return to running the main process normally...



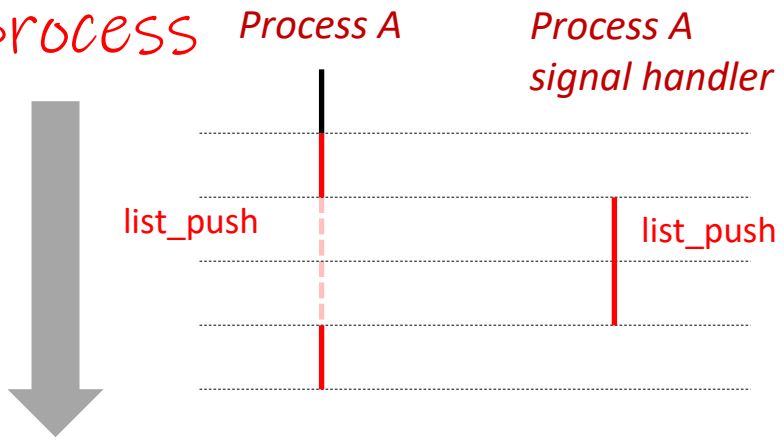
Critical Section Walkthrough

```
// 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

```c
// 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, NaN);
}

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?

Lecture Outline

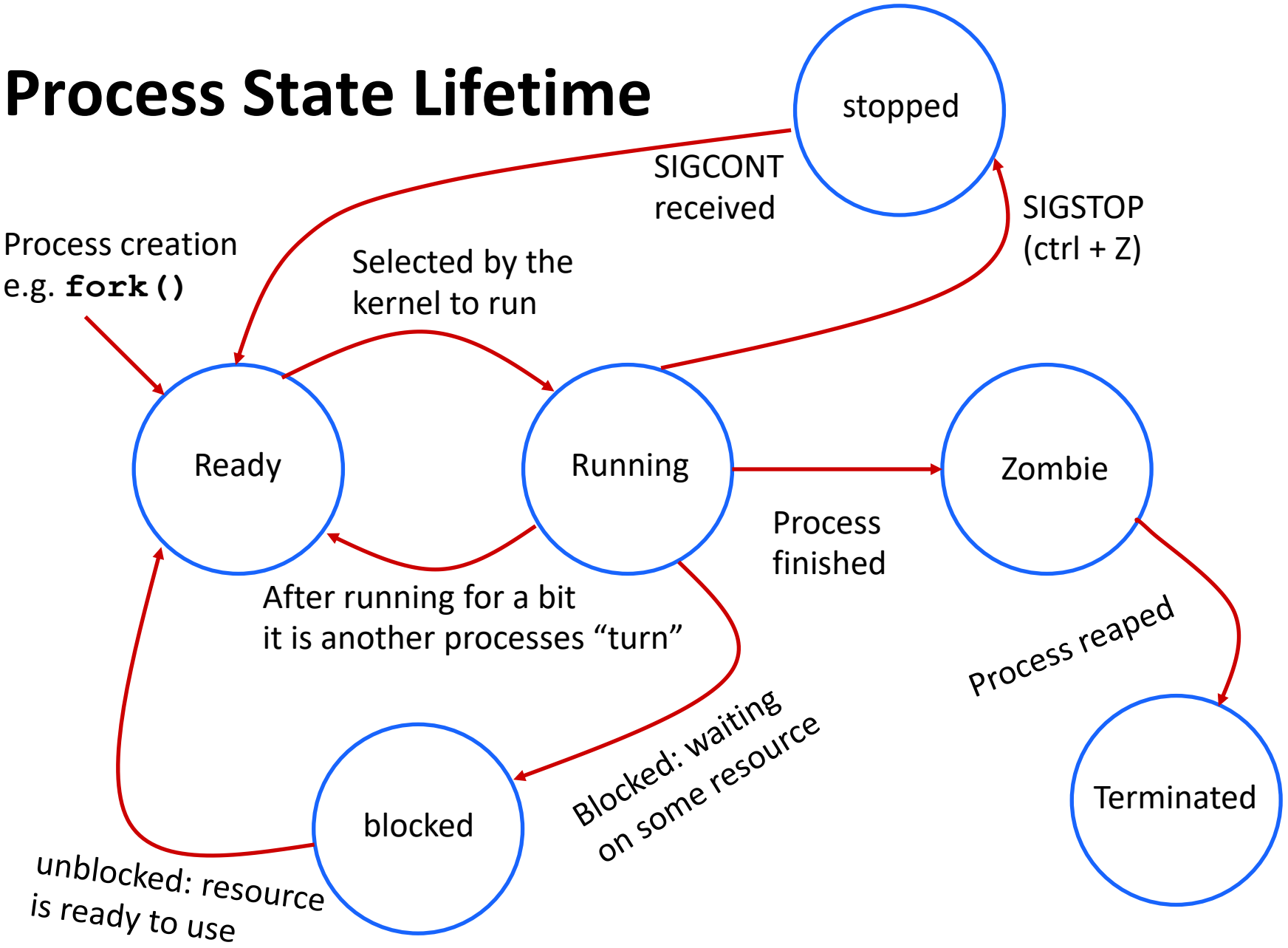
- Signal high level view
- Signal Blocking
  - `sigset_t` & `sigprocmask`
- Critical section & blocking
- **Updated process state diagrams: stop & continue**
- `sigsuspend` & busy waiting

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



Lecture Outline

- Signal high level view
- Signal Blocking
  - `sigset_t` & `sigprocmask`
- Critical section & blocking
- Updated process state diagrams: stop & continue
- **`sigsuspend` & busy waiting**

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've done this before, see `delay_sigint.c`

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

pollev.com/tqm

- Why might busy waiting be bad?
  It is not like the program can do anything else while it is waiting, so why is it bad?

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 ☺
  ```