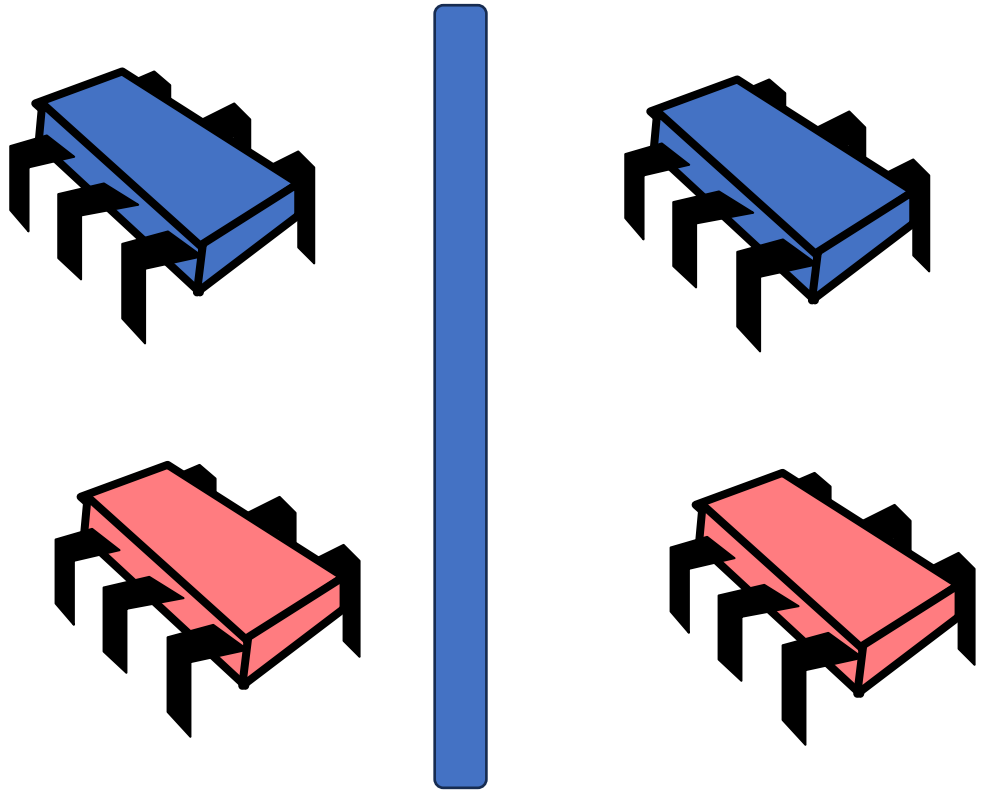
CSE113: Parallel Programming

- **Topics**:
  - Barriers
  - Processes



Announcements

- HW 4 grades will be released this week (after the holidays).

Announcements

- HW 5 is due this week on Thursday.

Announcements

SETs are out, please do them! It helps us out a lot.

Review

Barriers

Schedule

- **Barriers**
  - Specification
  - **Implementation**

Barrier Implementation

- First attempt at implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        // ??
     }

}
```

Barrier Implementation

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        // What next?
     }

}
```

Barrier Implementation

First handle the case where the thread is the last thread to arrive

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        // What next?
     }

}
```

Barrier Implementation

Spin while there is a thread waiting at the barrier

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

Barrier Implementation

Spin while there is a thread waiting at the barrier

Does this work?

```
class Barrier {
  private:
    atomic_int counter;
    int num_threads;
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
    }

     void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
     }

}
```

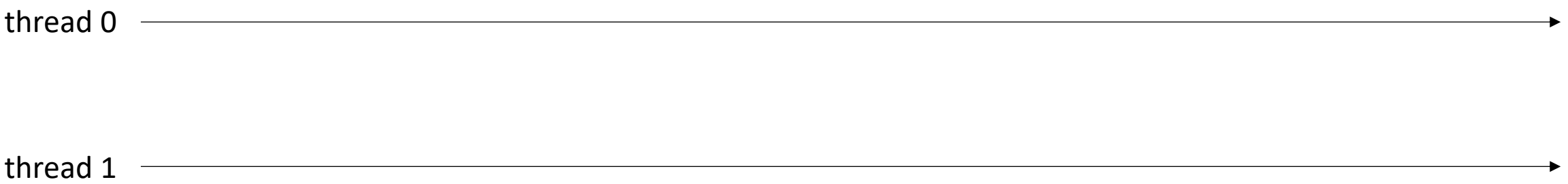
Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```



num_threads == 2

Thread 0:

```
B.barrier();
B.barrier();
```

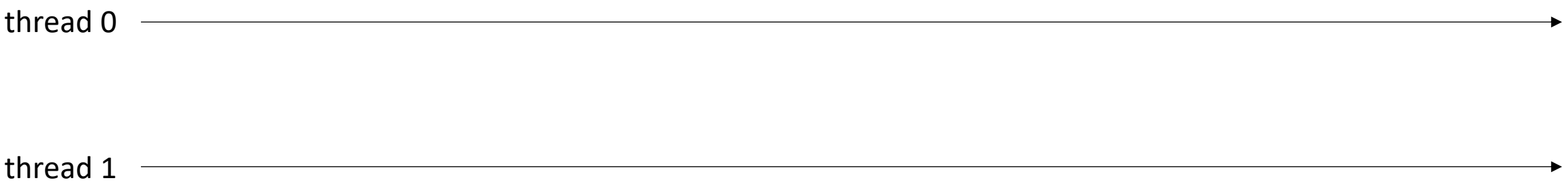
```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0



```
num_threads == 2
counter == 2
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0

thread 0 ⟶

thread 1 ⟶

```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
            counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num = 1

arrival_num = 0

thread 0 ⟶

thread 1 ⟶

num_threads == 2
counter == 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

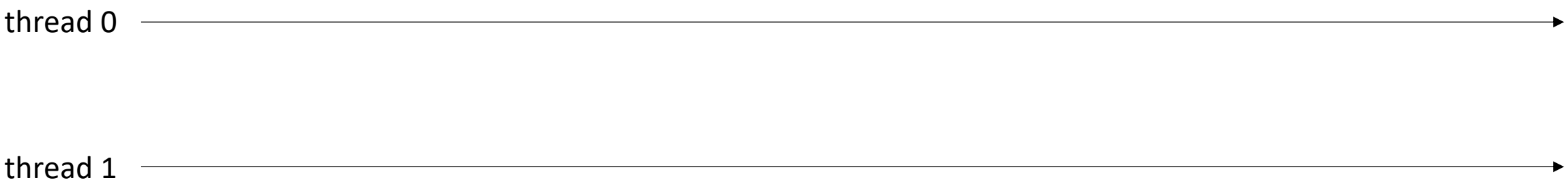
Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier



```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Leaves barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

```
num_threads == 2
counter == 0
```

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

enters next barrier

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads - 1) {
           counter.store(0);
        }
        else {
          while (counter.load() != 0);
        }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

but what if the OS preempted thread 1? Or it was asleep?

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 wakes up! Doesn't think its missed anything

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

num_threads == 2
counter == 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 wakes up! Doesn't think its missed anything

arrival_num == 0

arrival_num = 0

in a perfect world,
thread 1 executes now and leaves the barrier

Both threads get stuck here!

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B.barrier();
B.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Thread 0:

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

Thread 0:

```
B0.barrier();
B1.barrier();
```

```
void barrier() {
       int arrival_num = atomic_fetch_add(&counter, 1);
       if (arrival_num == num_threads - 1) {
          counter.store(0);
       }
       else {
         while (counter.load() != 0);
       }
    }
```

Thread 1:

```
B0.barrier();
B1.barrier();
```

Ideas for fixing?

Two different barriers that alternate?

Pros: simple to implement

Cons: user has to alternate barriers

```
B.barrier();
if (...) {
  B.barrier();
}
B.barrier();
```

How to alternate these calls?
Switching cannot be static,
has to be dynamic.

Sense Reversing Barrier

- Alternating “sense” dynamically

Thread 0:

```
B.barrier();
B.barrier();
```

sync on sense = false

Thread 1:

```
B.barrier();
B.barrier();
```

Sense Reversing Barrier

- Alternating “sense” dynamically

Thread 0:

```
B.barrier();
B.barrier();
```

sync on sense = true

Thread 1:

```
B.barrier();
B.barrier();
```

```
class SenseBarrier {
  private:
    atomic_int counter;
    int num_threads;
    atomic_bool sense;
    bool thread_sense[num_threads];
  public:
    Barrier(int num_threads) {
      counter = 0;
      this->num_threads = num_threads;
      sense = false;
      thread_sense = {true, ...};
    }

     void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
}
```

Set sense to what threads are waiting for

thread_sense = true

```
num_threads == 2
counter == 0
sense = false
```

thread_sense = true

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 2
sense = false

thread_sense = true
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 2
sense = false

thread_sense = true
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 0
sense = true

thread_sense = false
arrival_num = 1

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 0
sense = true

thread_sense = false
arrival_num = ?

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

Remember the issue! Thread 1 went to sleep around this time and thread 0 went into the barrier again!

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = true
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

both are waiting!,
but thread 1 can leave

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 0

Thread 1:

```
B.barrier();
B.barrier();
```

both are waiting!,
but thread 1 can leave

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = ?

Thread 1:

```
B.barrier();
B.barrier();
```

Thread 1 finishes the barrier

num_threads == 2
counter == 1
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = ?

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 2
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 2
sense = true

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 0
sense = false

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

Goes into the second barrier

num_threads == 2
counter == 0
sense = false

thread_sense = false
arrival_num = 0

Thread 0:

```
B.barrier();
B.barrier();
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense = thread_sense[tid];
        }
        else {
          while (sense != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
    }
```

thread_sense = false
arrival_num = 1

Thread 1:

```
B.barrier();
B.barrier();
```

thread 0 can leave, thread 1 can leave and the barrier works as expected!

Processes

Processes

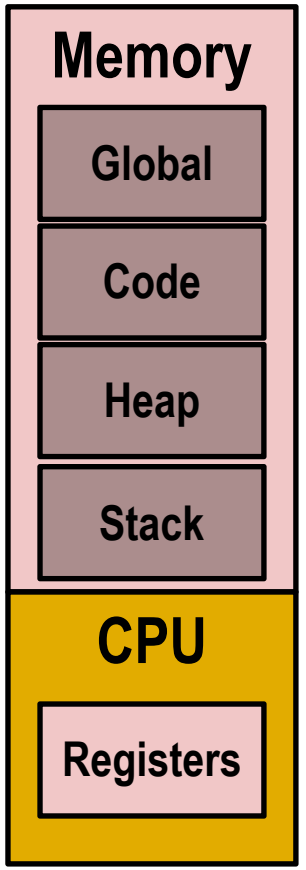
■**Definition: A *process* is an instance of a running program.**

■**Process provides each program with two key abstractions:**

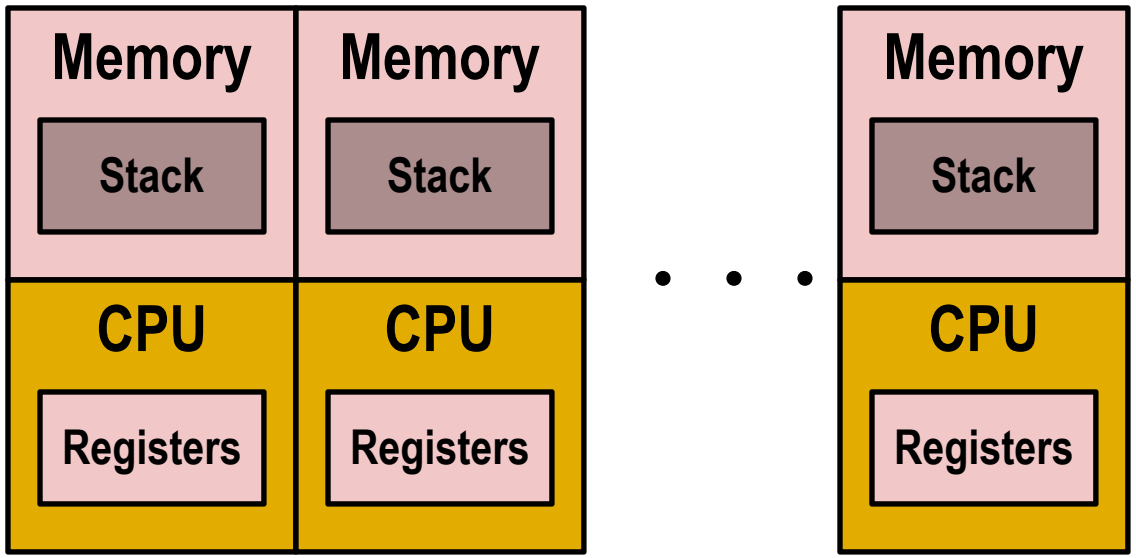
- ▪Logical control flow
  - ▪Each program seems to have exclusive use of the CPU
- ▪Private copy of program state
  - ▪Register values (PC, stack pointer, general registers, condition codes)
  - ▪Private virtual address space

    Program has exclusive access to main memory

    Including stack

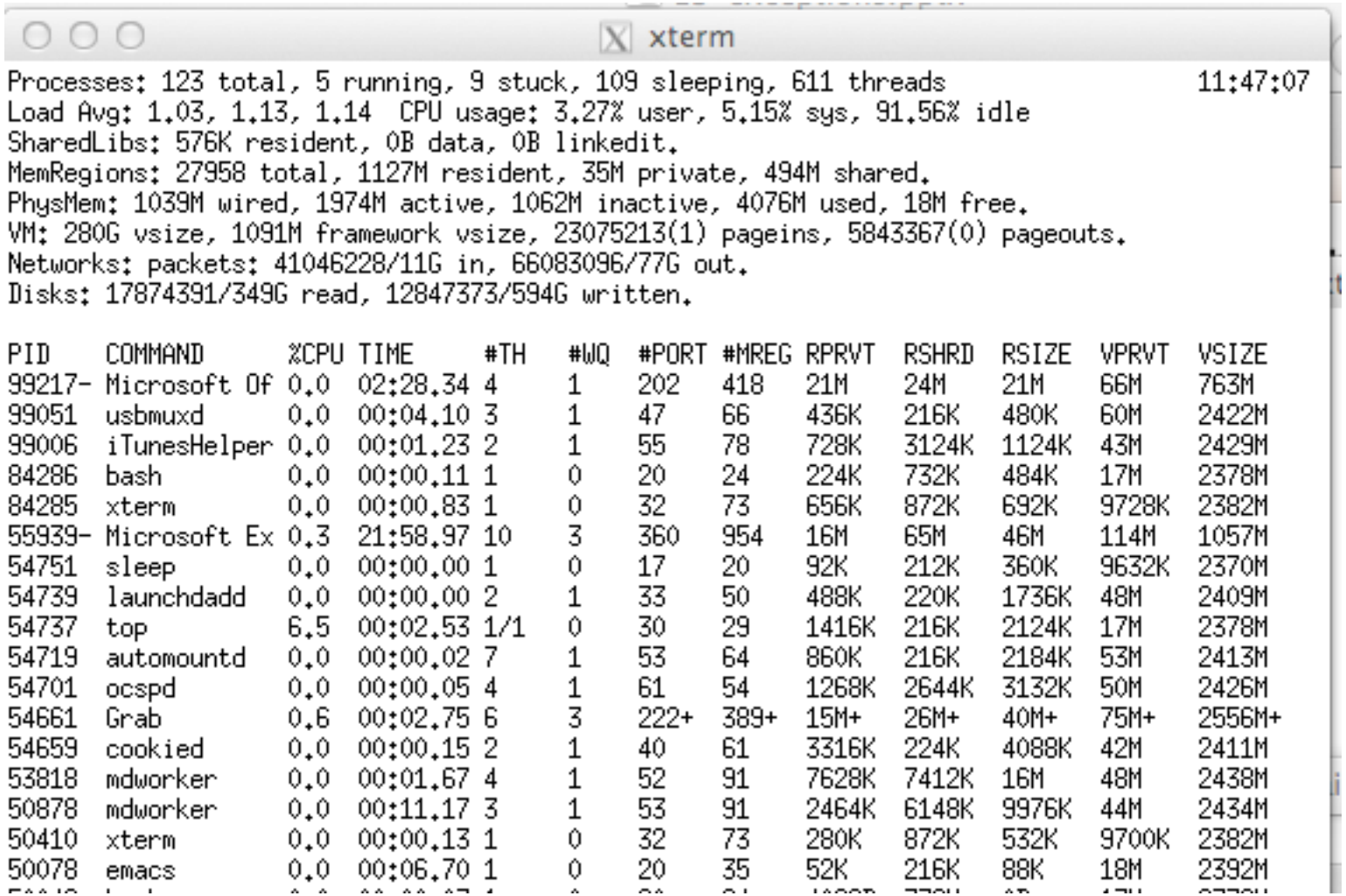


Multiprocessing: The Illusion



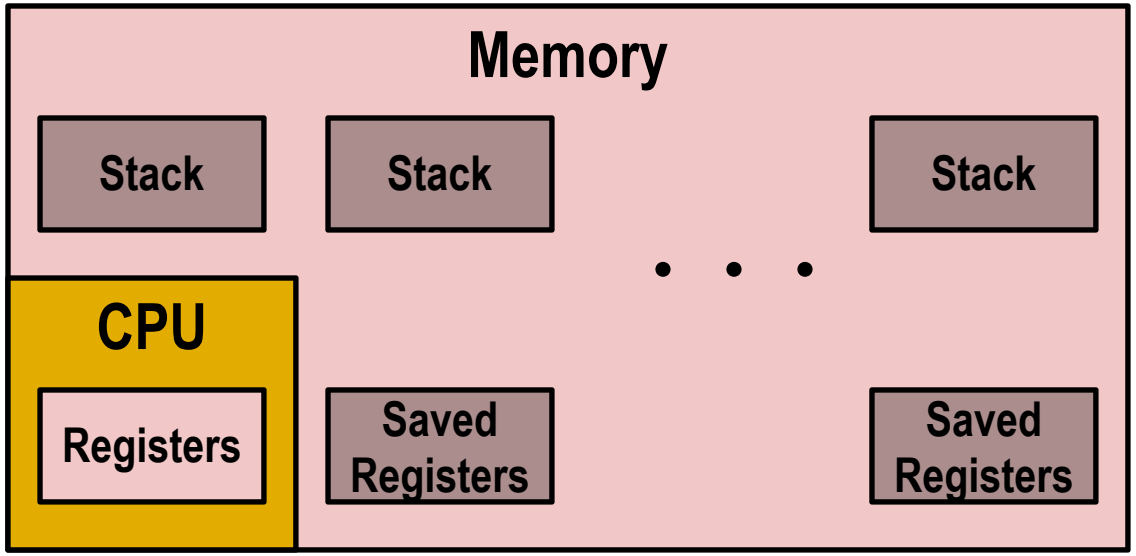
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example



- **Running program "top" on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality

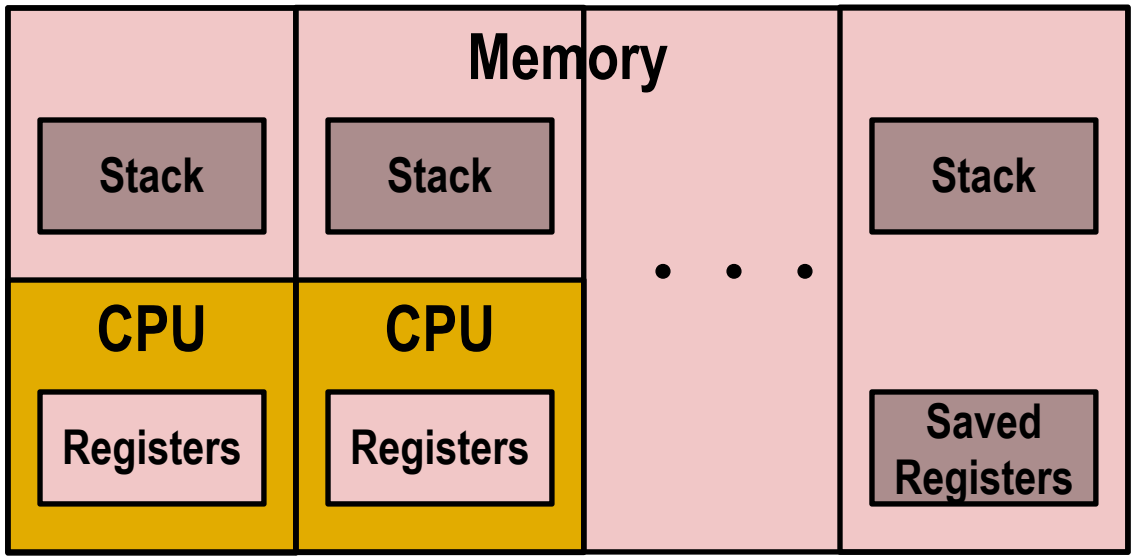


- **Single Processor Executes Multiple Processes Concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for non-executing processes saved in memory

The World of Multitasking

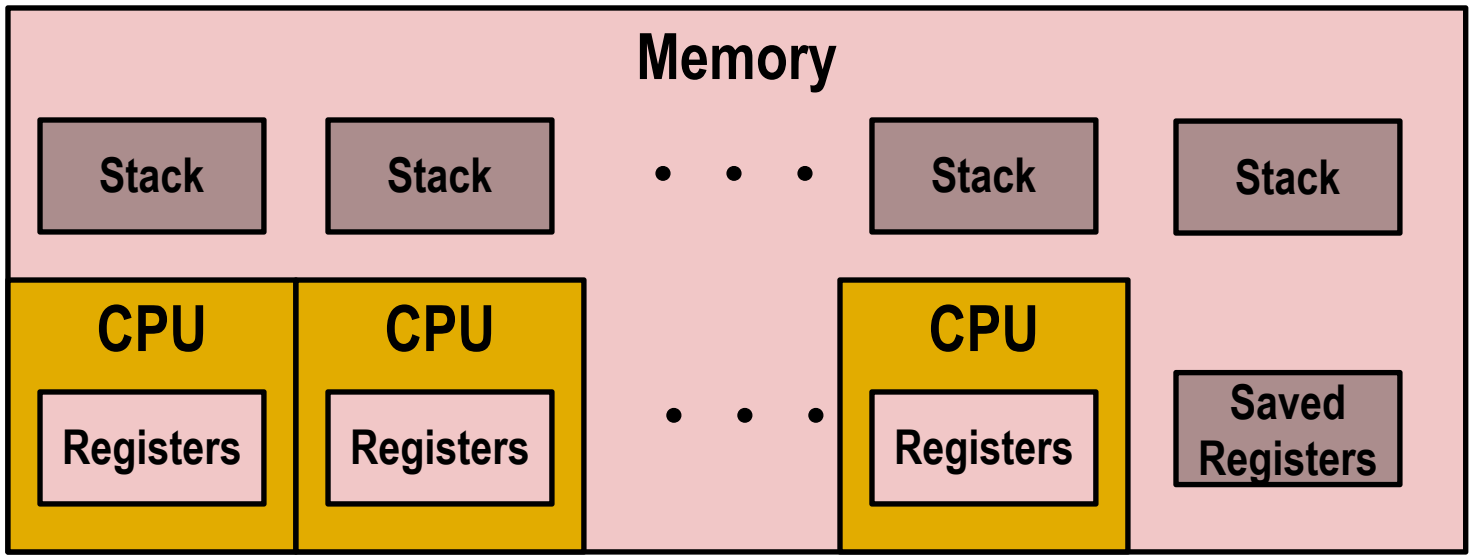
- **System runs many processes concurrently**
- **Regularly switches from one process to another**
  - Suspend process when it needs I/O resource or timer event occurs
  - Resume process when I/O available or given scheduling priority
- **Appears to user(s) as if all processes executing simultaneously**
  - Even though systems can only execute one process (or a small number of processes) at a time
  - Except possibly with lower performance than if running alone

Multiprocessing: The (New) Reality



- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processes onto cores done by OS

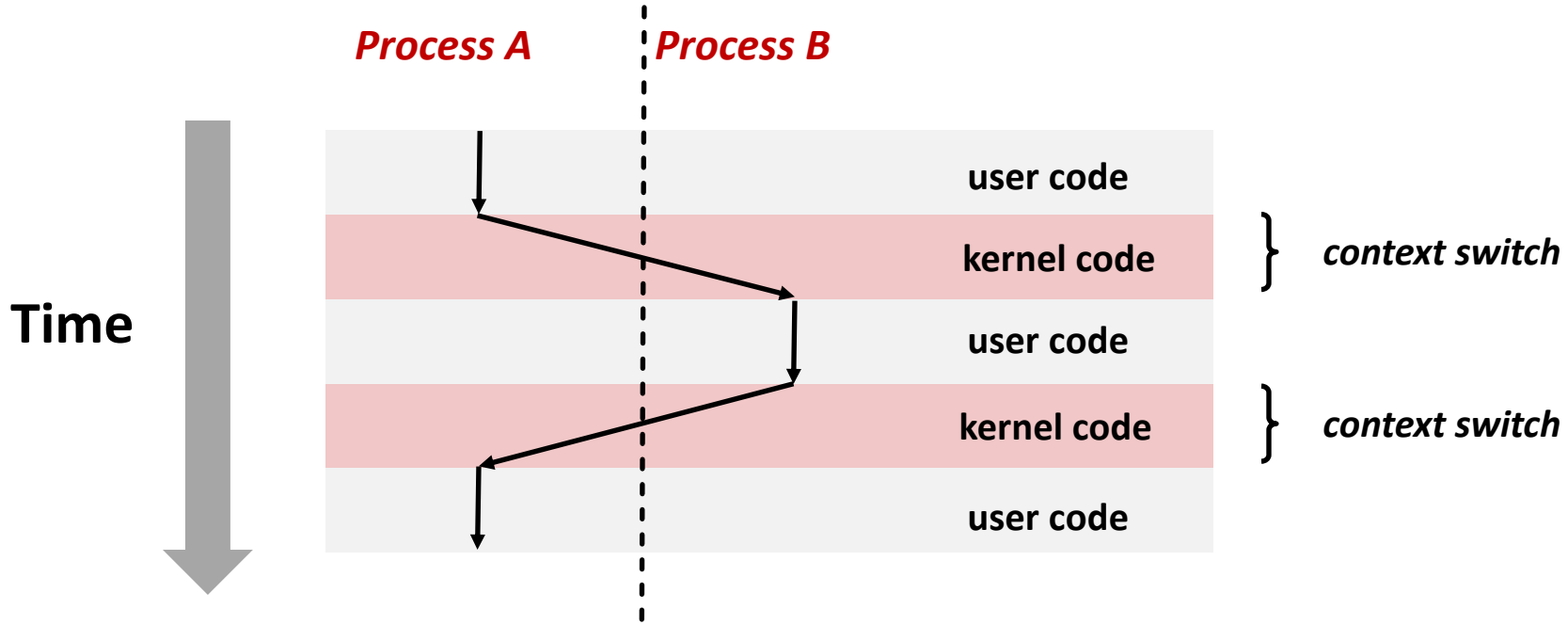
Multithreading: The Illusion



- **Single process runs multiple *threads* concurrently**
- **Each has own control flow and runtime state**
  - But view part of memory as shared among all threads
  - One thread can read/write the state of another

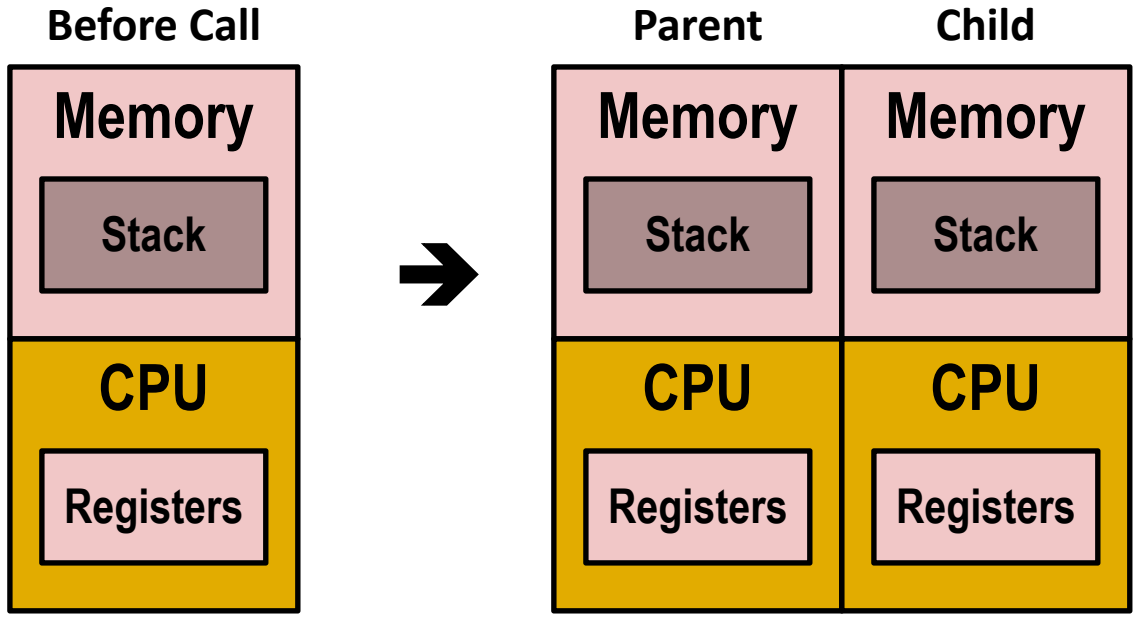
Context Switching

- Processes are managed by a shared chunk of OS code
- called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch*



`fork`: Creating New Processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - (Appears to) create complete new copy of program state
  - Child & parent then execute as independent processes
    - Writes by one don't affect reads by other
    - But … share any open files



`fork`: Details

- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's **`pid`** (process id) to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- **Fork is interesting (and often confusing) because**
- **it is called *once* but returns *twice***

Understanding fork

Process n

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Child Process m

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

pid = 0

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

`hello from parent` *Which one is first?* `hello from child`

Fork Example #1

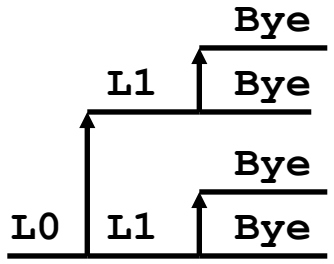
- **Parent and child both run the same code**
  - Distinguish parent from child by return value from **`fork`**
- **Start with same state, but each has private copy**
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Fork Example #2

- Two consecutive forks

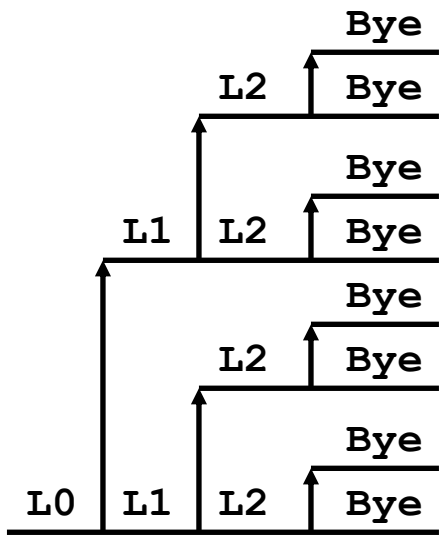
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Three consecutive forks

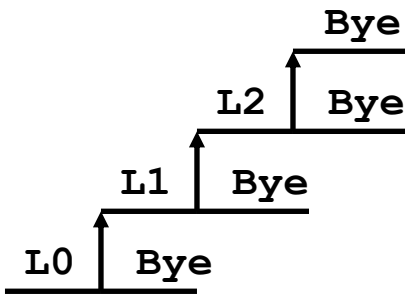
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



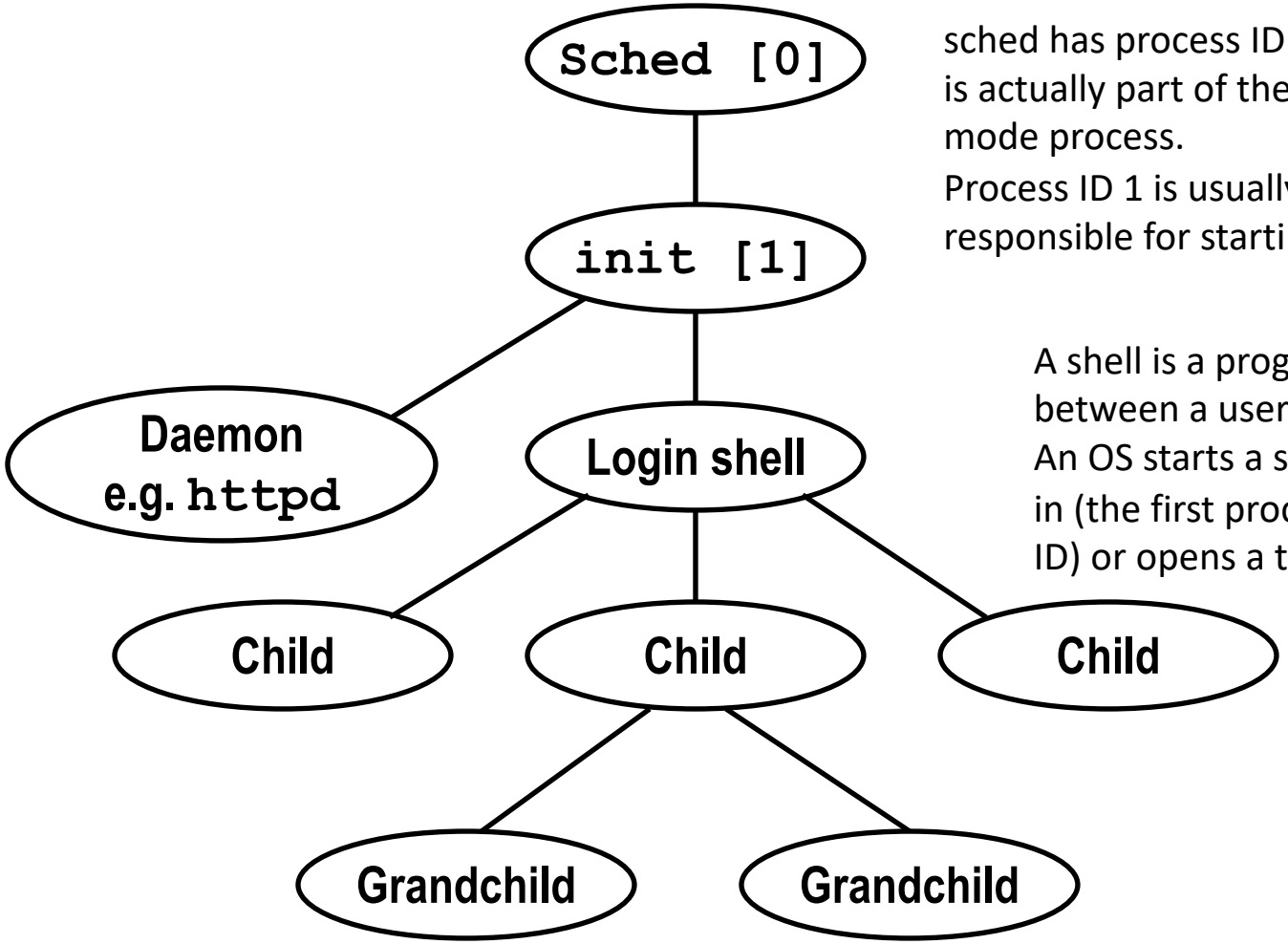
Fork Example #4

- Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Unix Process Hierarchy



sched has process ID 0 and is responsible for paging, and is actually part of the kernel rather than a normal user-mode process.
Process ID 1 is usually the init process primarily responsible for starting and shutting down the system.

A daemon is a computer program that runs as a background process, rather than being under the direct control of an interactive "user". Traditionally, the process names of a daemon end with the letter d

A shell is a program that provides an interface between a user and an operating system (OS) kernel. An OS starts a shell for each user when the user logs in (the first process that executes under your user ID) or opens a terminal or console window.

`exit`: Ending a process

- **`void exit(int status)`**
  - exits a process
    - Normally return with status 0
  - **`atexit()`** registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

- **Idea**
  - When process terminates, still consumes system resources
    - Various tables maintained by OS
  - Called a “zombie”
    - Living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel discards process
- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **ps** shows child process as "defunct"
- Killing parent allows child to be reaped by **init**

Orphan process: Nonterminating Child process

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated. The process init adopts the process. Daemons can be created this way.

- Must kill explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

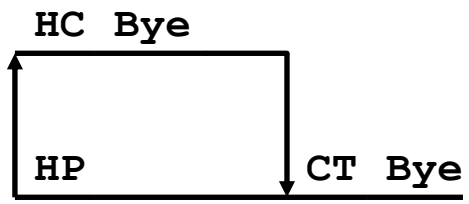
- **Parent reaps child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - suspends current process until one of its children terminates
  - return value is the **`pid`** of the child process that terminated
  - if **`child_status != NULL`**, then the object it points to will be set to a status indicating why the child process terminated

`wait`: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status (W for wait)

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

`waitpid()`: Waiting for a Specific Process

- **`waitpid(pid, &status, options)`**
  - suspends current process until specific process terminates
  - various options

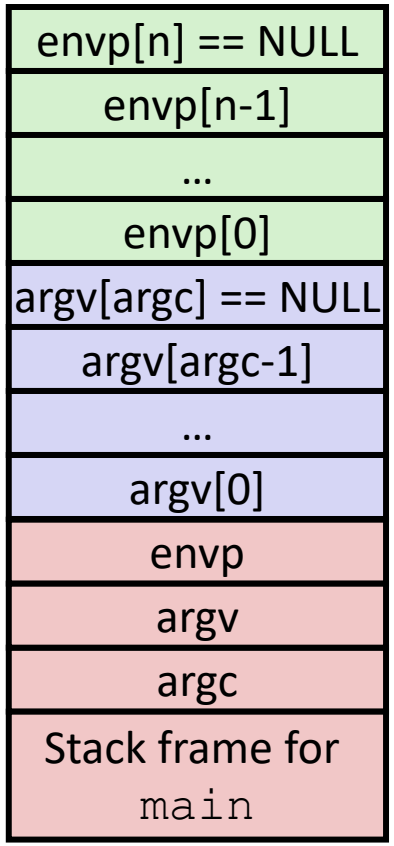
```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

`execve`: Loading and Running Programs

```
int execve(
   char *filename,
   char *argv[],
   char *envp[]
)
```

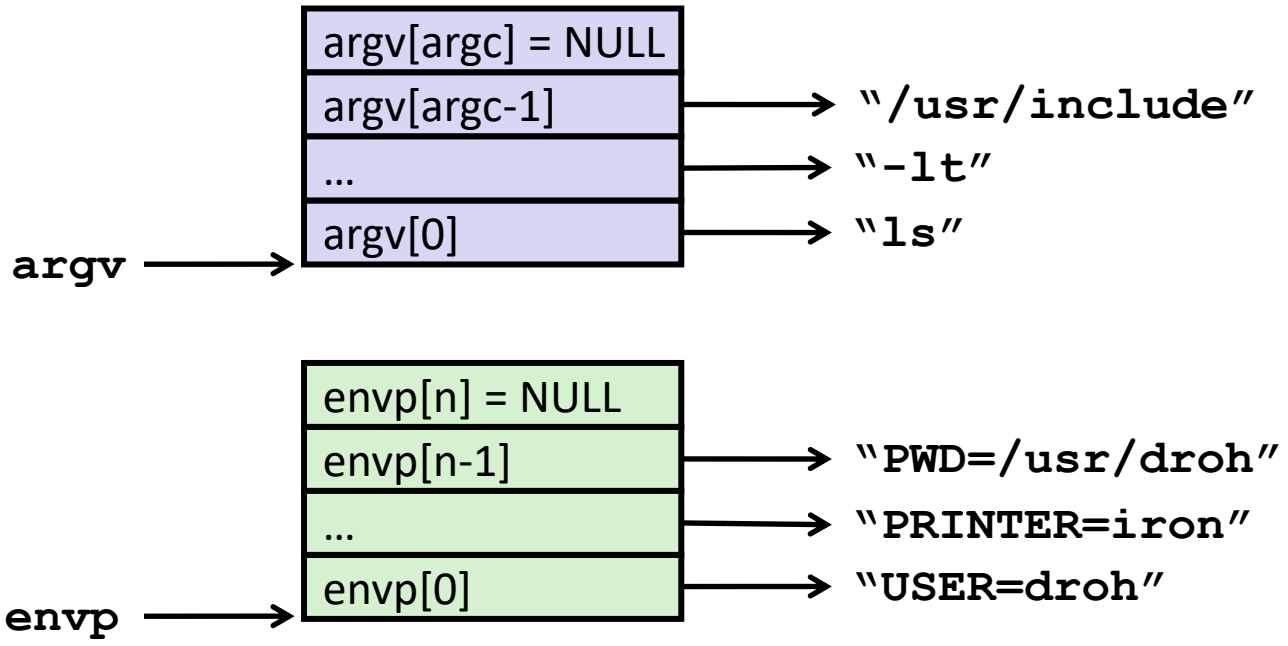
- **Loads and runs in current process:**
  - Executable **`filename`**
  - With argument list **`argv`**
  - And environment variable list **`envp`**
- **Does not return (unless error)**
- **Overwrites code, data, and stack**
  - keeps pid, open files
- **Environment variables:**
  - "name=value" strings
  - Use functions `getenv` and `putenv` to access environment variables.

The v and e comes from the fact that it takes an argument argv, envp to the vector of arguments and environment variables to the program



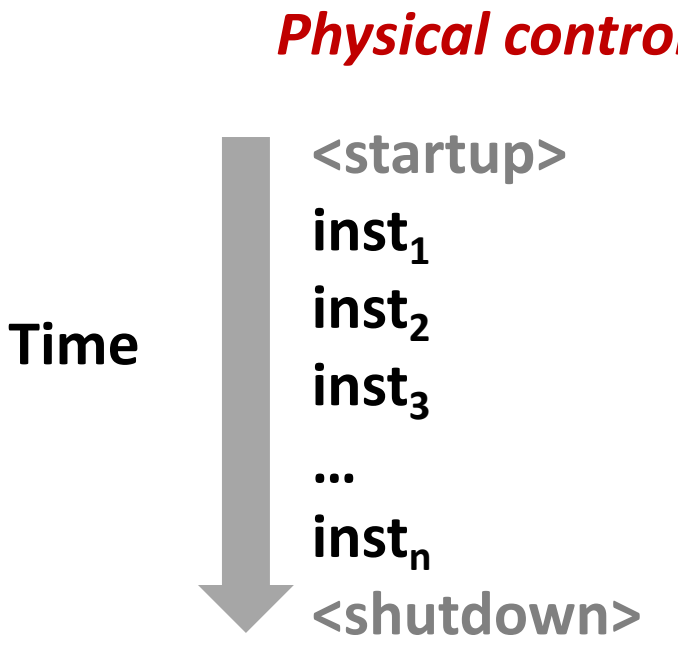
execve Example

```
if ((pid = fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, envp) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```



Control Flow

- **Processors do only one thing:**
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)



Altering the Control Flow

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return

Both react to changes in ***program state***

- **Insufficient for a useful system:**
- **Difficult to react to changes in *system state***
  - data arrives from a disk or a network adapter
  - user hits Ctrl-C at the keyboard
  - System timer expires
  - instruction divides by zero

- **System needs mechanisms for “exceptional control flow”**

Exceptional Control Flow

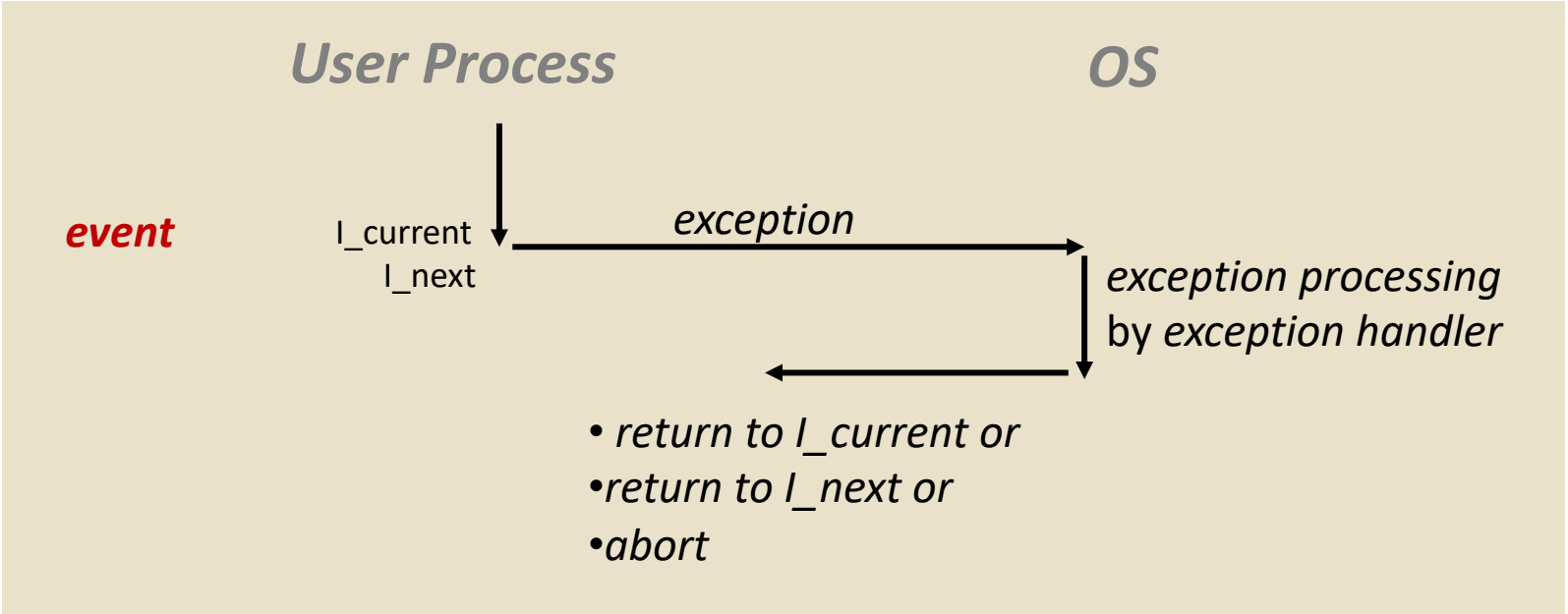
Exists at all levels of a computer system:

- **Low level mechanisms**
  - Exceptions
    - Events external to the CPU, or
    - abnormal execution of an instruction inside the CPU
    - Implemented via combination of hardware and OS kernel software

- **Higher level mechanisms**
  - Process context switch
    - Hardware timer and OS kernel software
  - Signals
    - OS kernel software
  - Nonlocal jumps: setjmp()/longjmp()
    - C language runtime library (nonlocal jumps)

Exceptions

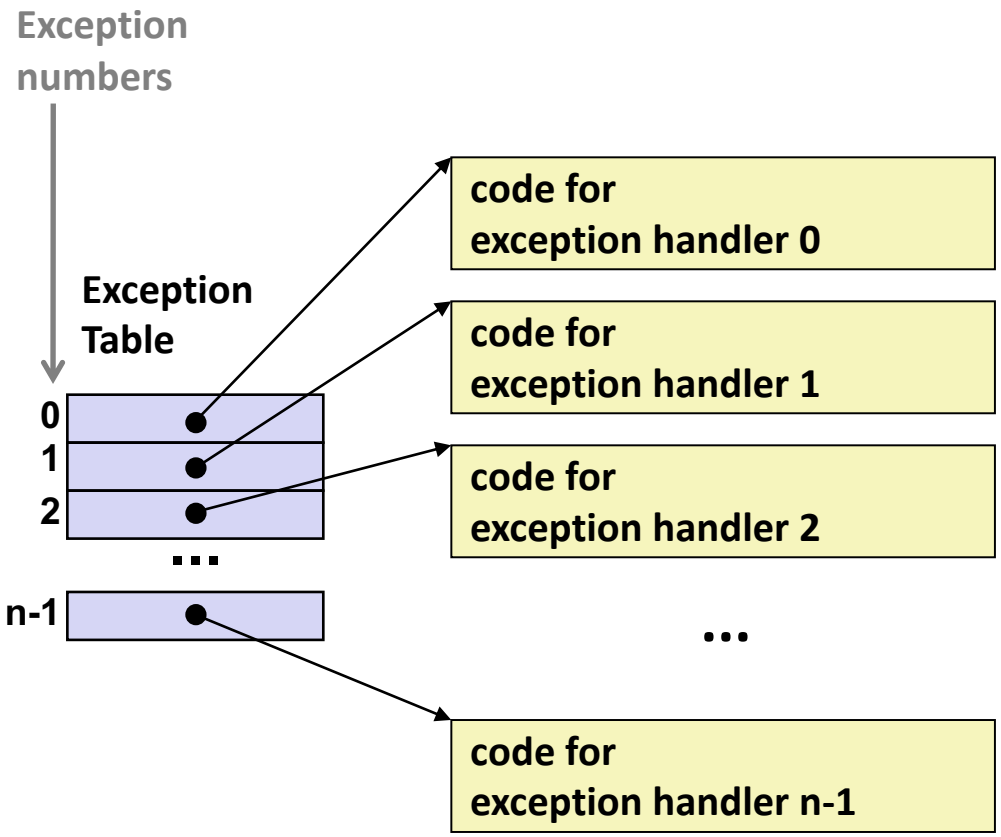
■An ***exception*** **is a transfer of control to the OS in response to some *event* (i.e., change in processor state)**



■**Examples:**

■div by 0, arithmetic overflow, page fault, page fault, Ctrl-C

Exception Tables



- Each type of event has a
- unique exception number k

- k = index into exception table
- (a.k.a. interrupt vector)

- Handler k is called each time
- exception k occurs

Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction

- **Examples:**
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the power button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC

Synchronous Exceptions

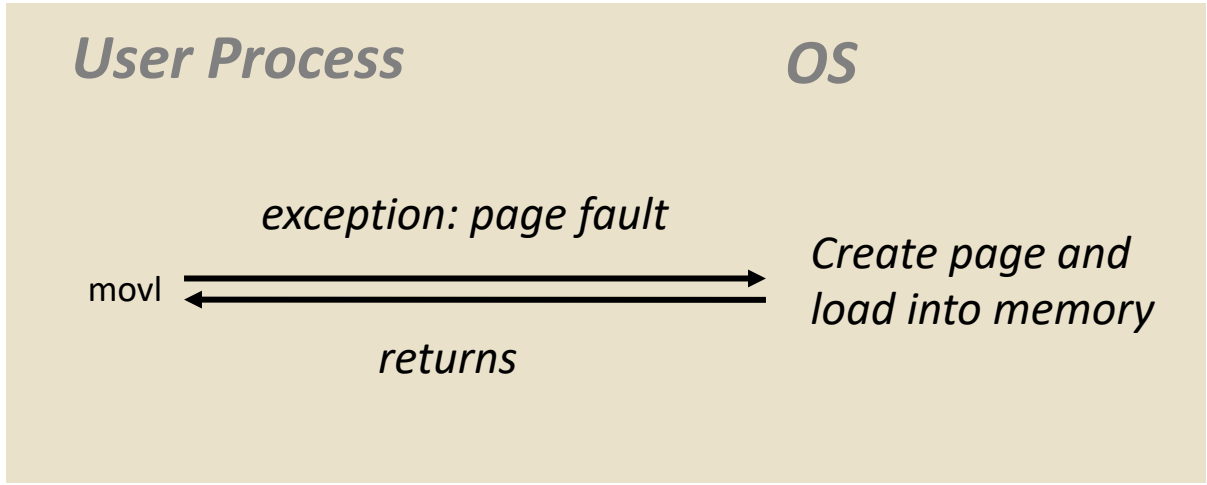
- **Caused by events that occur as a result of executing an instruction:**
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("**current**") instruction **or aborts**
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: parity error, machine check
    - **Aborts** current program
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps
    - Returns control to "**next**" instruction

Fault Example: Page Fault

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

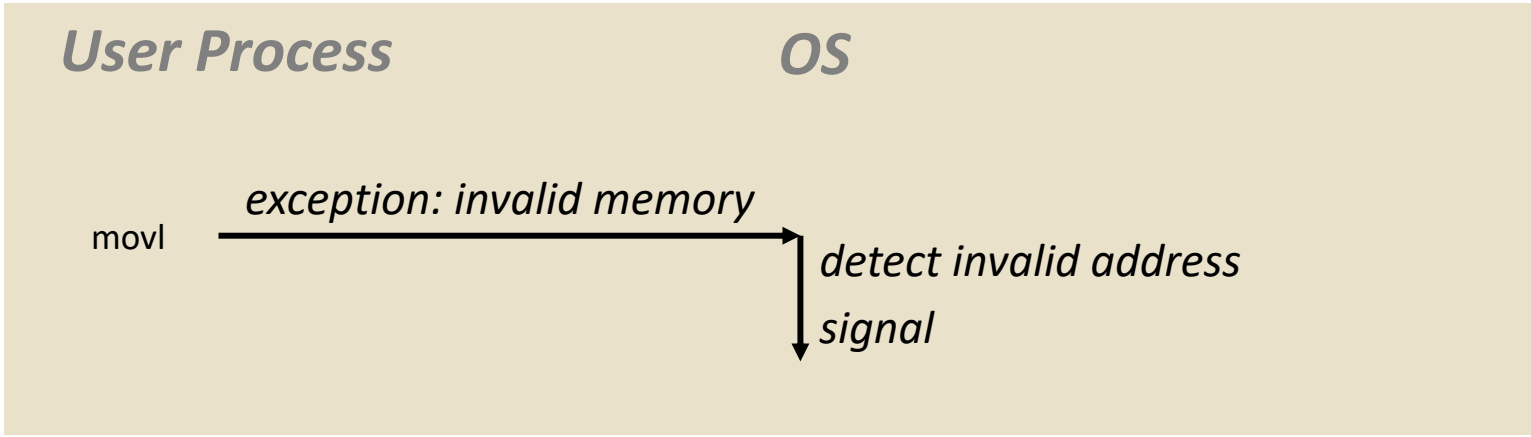


- Page handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
 80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```

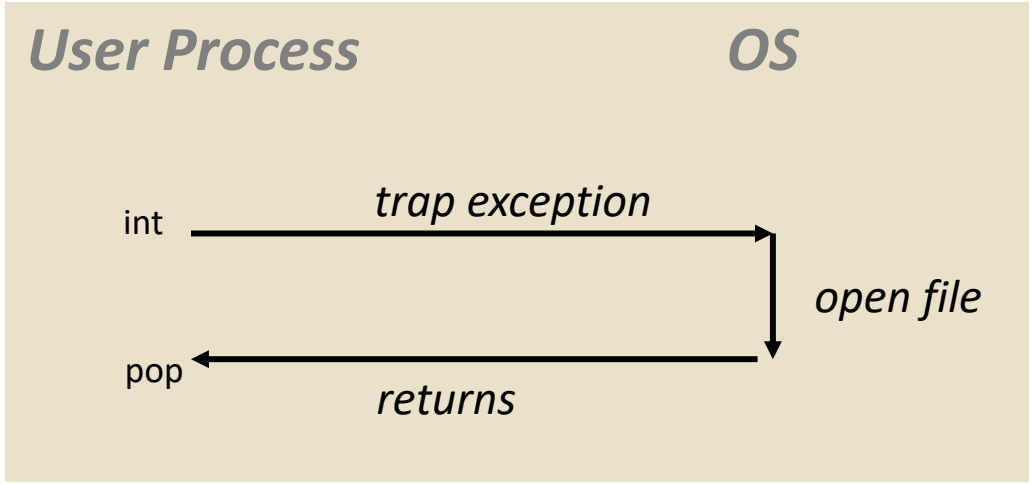


- Page handler detects invalid address
- Sends **`SIGSEGV`** (segmentation violation) signal to user process
- User process exits with “segmentation fault”

Trap Example: Opening File

- User calls: **`open(filename, options)`**
- Function **`open`** executes system call instruction **`int`**

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                    int    $0x80
 804d084:       5b                       pop    %ebx
 . . .
```



- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

Exception Table i386 (Intel Architecture, 32-bit)

Exception Number	*Description*	*Exception Class*
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-127	OS-defined	Interrupt or trap
128 (0x80)	System call	Trap
129-255	OS-defined	Interrupt or trap

General protection fault: accessing memory that it should not access. Attempting to write to a read-only portion of memory. Attempting to execute bytes in memory which are not designated as instructions. Attempting to read as data bytes in memory which are designated as instructions.

Exceptional Control Flow

Exists at all levels of a computer system:

- **Low level mechanisms**
  - Exceptions
    - Events external to the CPU, or
    - abnormal execution of an instruction inside the CPU
    - Implemented via combination of hardware and OS kernel software

- **Higher level mechanisms**
  - Process context switch
    - Hardware timer and OS kernel software
  - Signals
    - OS kernel software and application software
  - Nonlocal jumps: setjmp()/longjmp()
    - C language runtime library (nonlocal jumps)

Shell Programs

■**A *shell* is an application program that runs programs on behalf of the user.**

- **`sh`** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
- **`csh`** BSD Unix C shell (**`tcsh`**: enhanced `csh` at CMU and elsewhere)
- **`bash`** "Bourne-Again" Shell

```
int main() {
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

Execution is a sequence of read/evaluate steps

What Is a “Background Job”?

- **Users generally run one command at a time**
  - Type command, read output, type another command
- **Some programs run “for a long time”**
  - Example: “delete this file in two hours”
- **A “background” job is a process we don't want to wait for**

```
unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
```

```
unix> (sleep 7200 ; rm /tmp/junk) &
[1] 907
unix> # ready for next command
```

Simple Shell `eval` Function

```
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if ((pid = fork()) == 0) {   /* child runs user job */
        if (execve(argv[0], argv, environ) < 0) {
            printf("%s: Command not found.\n", argv[0]);
            exit(0);
        }
    }

    if (!bg) {   /* parent waits for fg job to terminate */
        int status;
        if (waitpid(pid, &status, 0) < 0)
            unix_error("waitfg: waitpid error");
    } else         /* otherwise, don't wait for bg job */
        printf("%d %s", pid, cmdline);
}
```

Problem with Simple Shell Example

- **Our example shell correctly waits for and reaps foreground jobs**

- **But what about background jobs?**
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory
  - Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1

```
unix> limit maxproc        # csh syntax
maxproc      202752
unix> ulimit -u            # bash syntax
202752
```

Signals to the Rescue!

- **Problem: Finished background jobs**
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow can't reap exited background processes in a timely fashion. Regular control flow is “wait until running job completes, then reap it”

- **Solution: Signal**
  - The kernel will interrupt regular processing to alert us when a background process completes

Signals

■A *signal* is a small message that notifies a process that an event of some type has occurred in the system

- akin to exceptions and interrupts
- sent from the kernel (sometimes at the request of another process) to a process
- signal type is identified by small integer ID's (1-30)
- Kernel delivers a signal to a *destination process* by updating some state in the context of the destination process
- only information in a signal is its ID and the fact that it arrived.

Sending a Signal

ID	*Name*	*Default Action*	*Corresponding Event*
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
8	SIGFPE	Terminate & Dump	Erroneous arithmetic operation
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

- SIGFPE: The SIGFPE signal is sent to a process when it executes an erroneous arithmetic operation. This may include integer division by zero, and integer overflow in the result of a divide.
- SIGKILL: Another process has invoked the **`kill`** system call to explicitly request the kernel to send a signal to the destination process
- SIGCHLD: the termination of a child process

Pending Signals

- **A signal is *pending* if sent but not yet received**
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **Kernel maintains `pending` bit vectors in the context of each process**

  `pending`: represents the set of pending signals
    - Kernel sets bit k in `pending` when a signal of type k is delivered
    - Kernel clears bit k in `pending` when a signal of type k is received

- **A pending signal is received at most once**

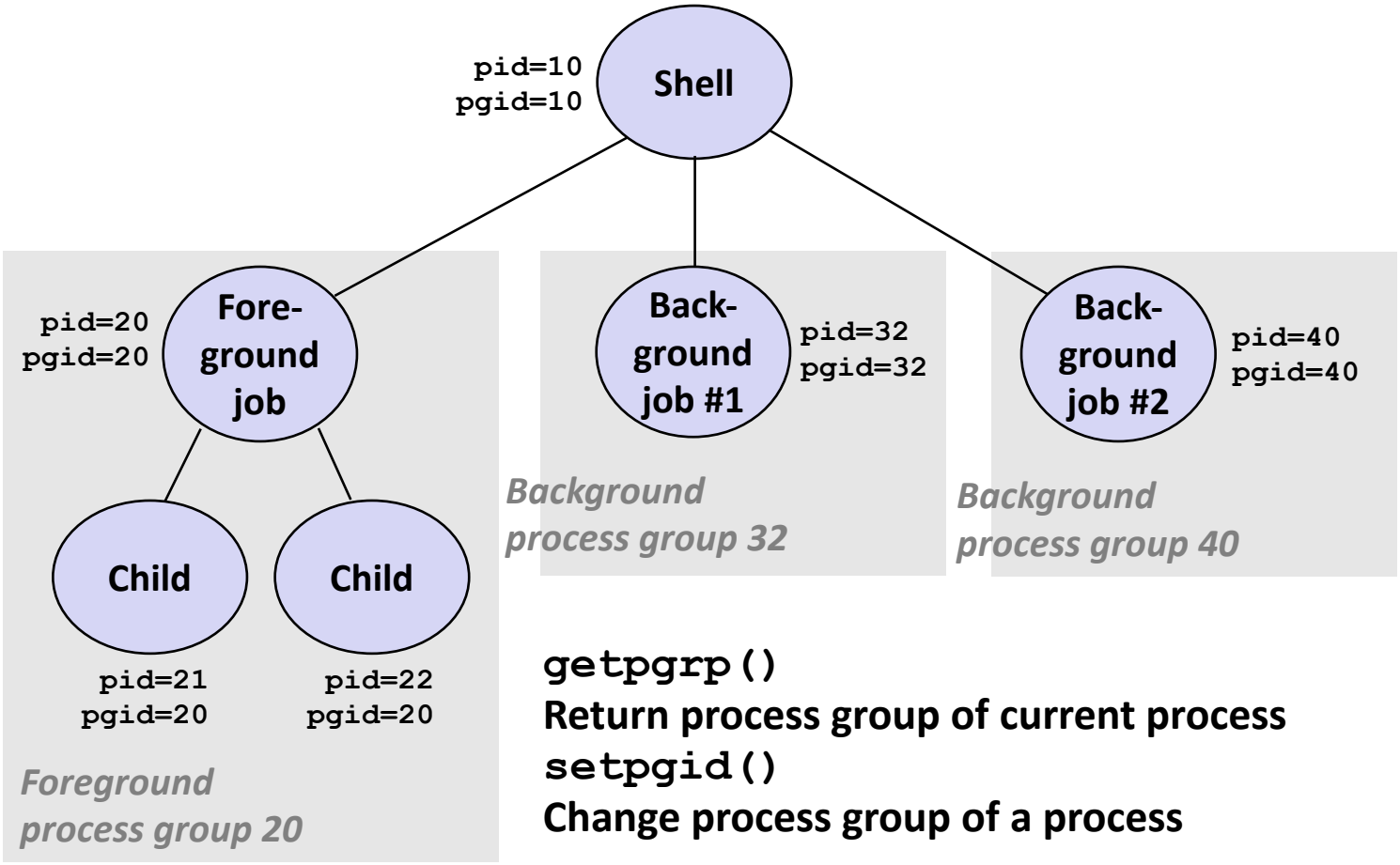
Receiving a Signal

- **A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal**

- **Three possible ways to react:**
  - ***Ignore*** the signal (do nothing)
  - ***Terminate*** the process (with optional core dump)
  - ***Catch*** the signal by executing a user-level function called ***signal handler***
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt

Process Groups

- Every process belongs to exactly one process group



`getpgrp()`
Return process group of current process
`setpgid()`
Change process group of a process

Sending Signals with /bin/kill Program

- **/bin/kill program sends arbitrary signal to a process or process group**

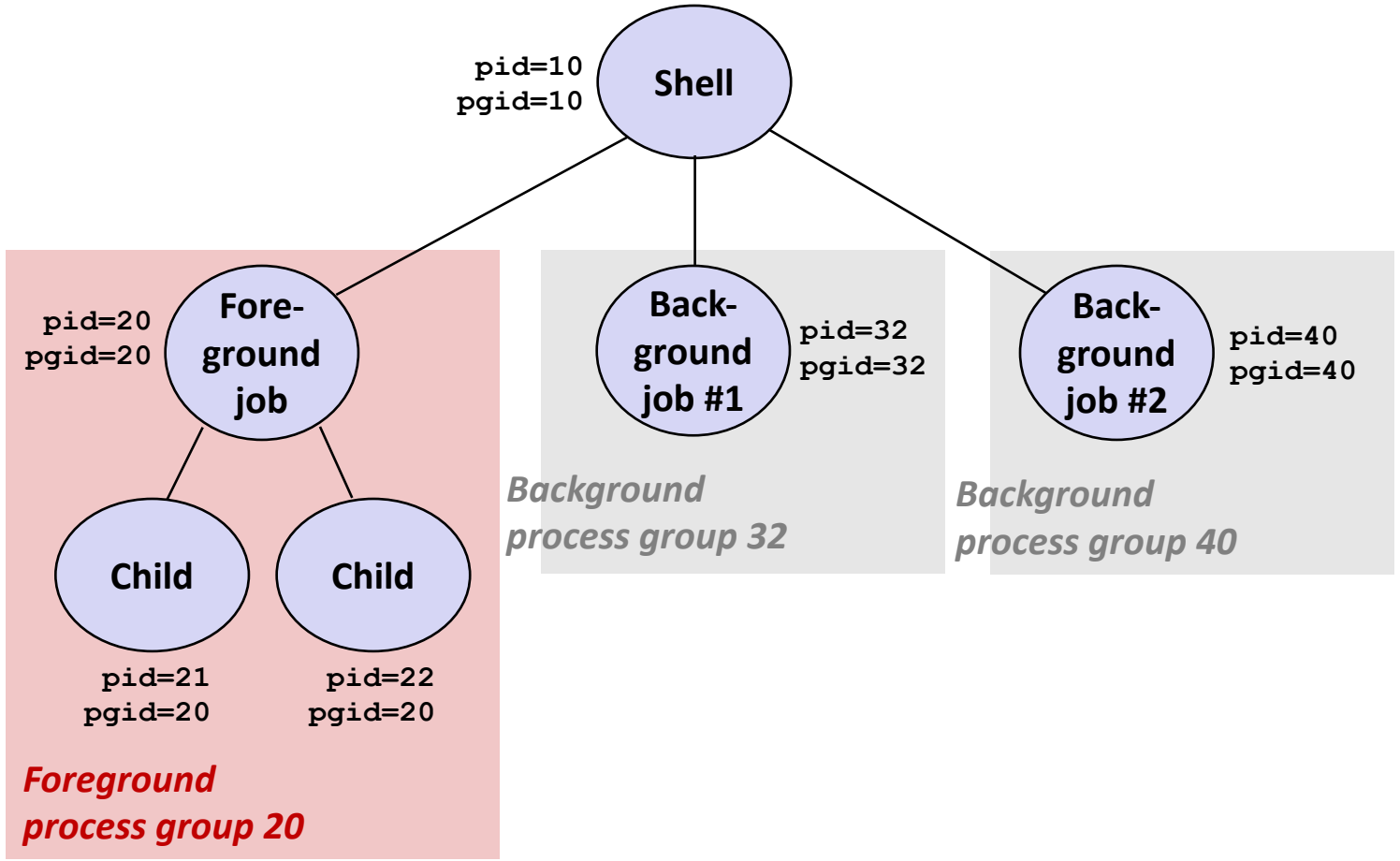
- **Examples**
  - **/bin/kill -9 24818**
    - Send SIGKILL to process 24818
  - **/bin/kill -9 -24817**
    - Send SIGKILL to every process in process group 24817
  - kill -TERM -- -5112
  - kill -9 -- -5112

```
linux> ./forks
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Sending Signals from the Keyboard

- **Typing ctrl-c (ctrl-z) sends a SIGINT (SIGTSTP) to every job in the foreground process group.**
  - SIGINT – default action is to terminate each process
  - SIGTSTP – default action is to stop (suspend) each process



Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./forks 17
28108 pts/8    T      0:01 ./forks 17
28109 pts/8    R+     0:00 ps w
bluefish> fg
./forks
<types ctrl-c>
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

STAT (process state) Legend:

First letter:

S: sleeping
T: stopped
R: running

Second letter:

s: session leader
+: foreground proc group

See “man ps” for more details

Sending Signals with `kill` Function

```
void fork12()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

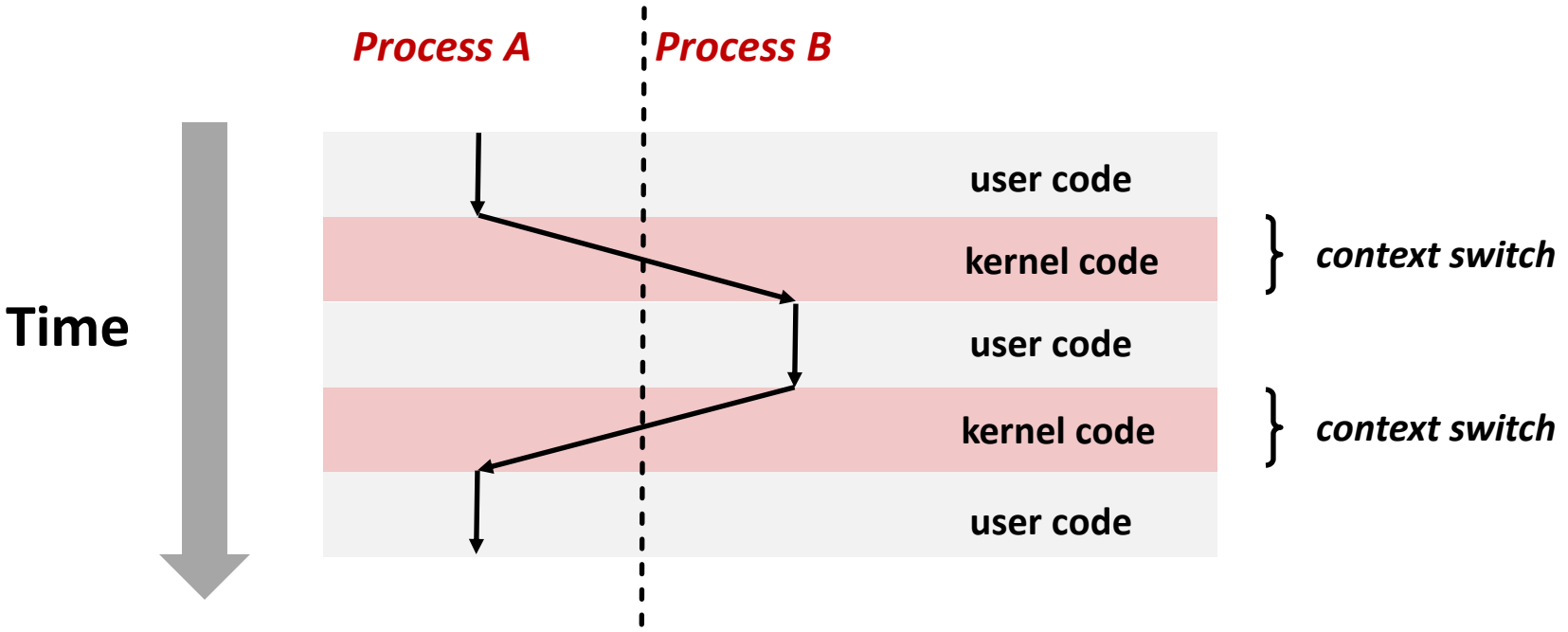
Blocked Signals

- **A process can *block* the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

- **Kernel maintains `pending` and `blocked` bit vectors in the context of each process**
  - **`blocked`**: represents the set of blocked signals
    - Can be set and cleared by using the **`sigprocmask`** function

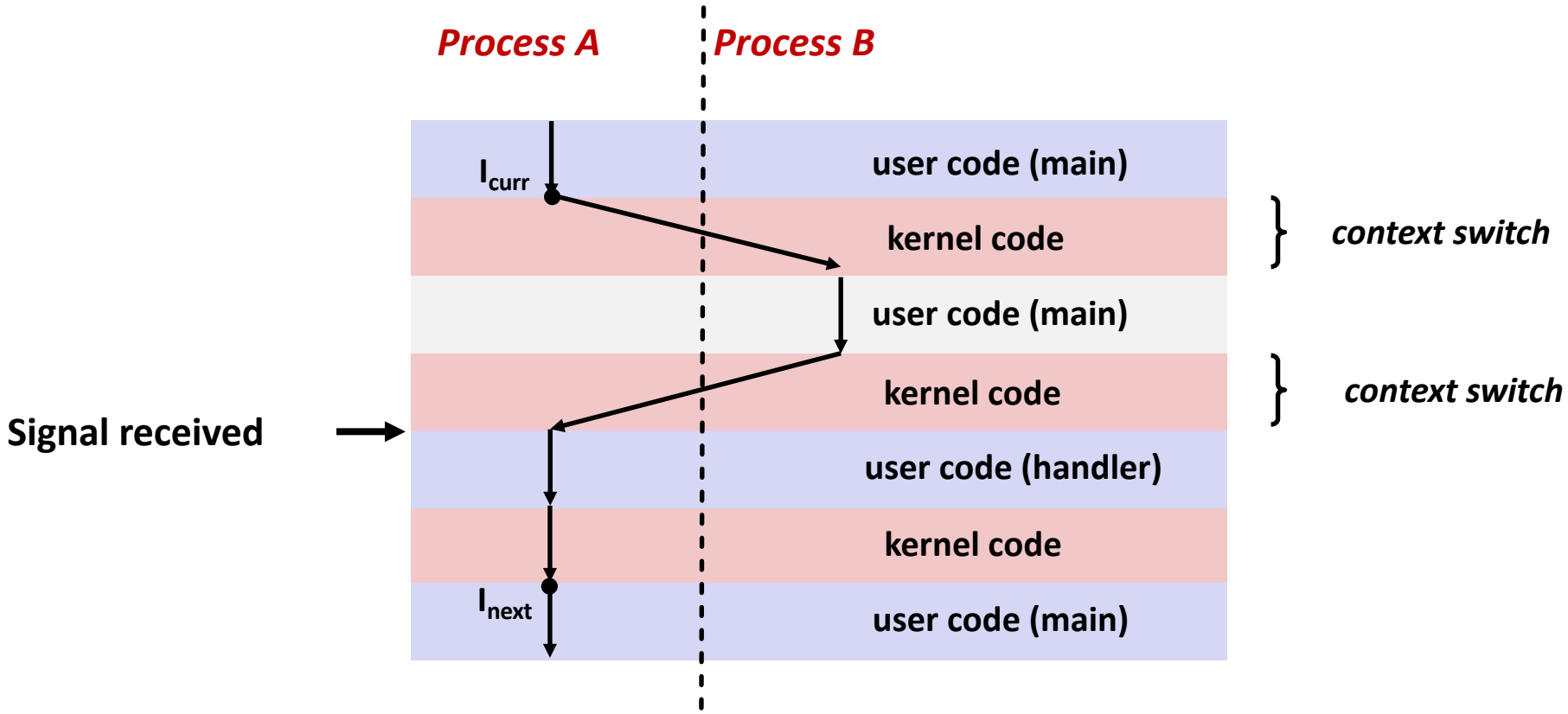
Receiving Signals

■Suppose kernel is returning from an exception handler and is ready to pass control to process *p*



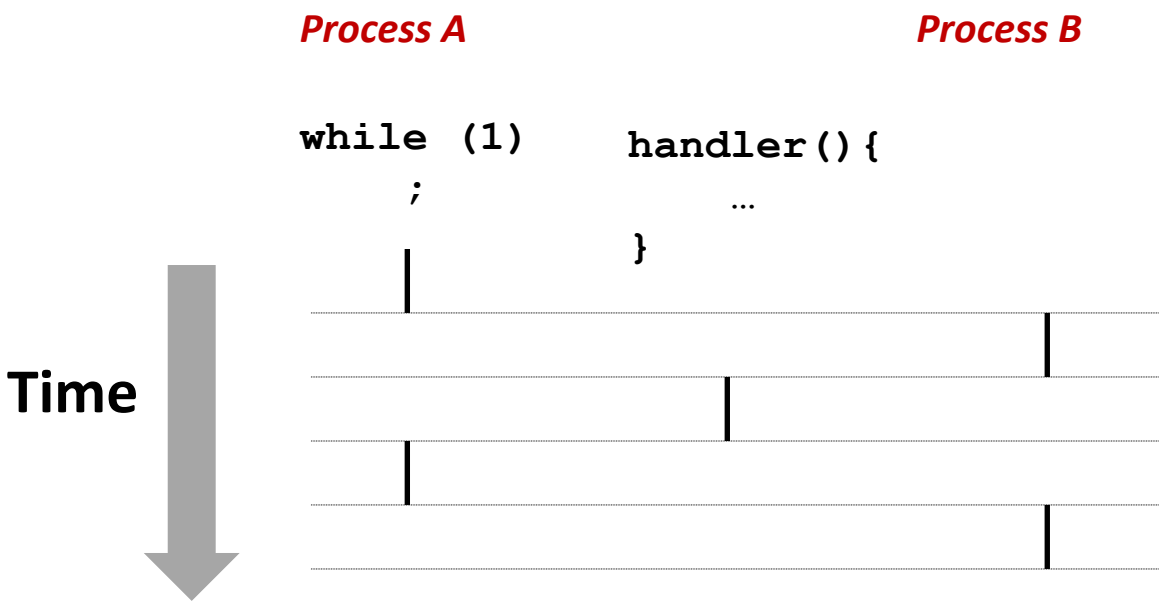
All context switches are initiated by calling some exceptional hander.

Receiving Signals



Signals Handlers as Concurrent Flows

■A signal handler is a separate logical flow (not another process) that runs concurrently with the main program



Receiving Signals

- **Suppose kernel is returning from an exception handler and is ready to pass control to process *p***

- **Kernel computes `pnb = pending & ~blocked`**
  - The set of pending nonblocked signals for process *p*

- **If (`pnb == 0`)**
  - Pass control to next instruction in the logical flow for *p*
- **Else**
  - Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*
  - The receipt of the signal triggers some ***action*** by *p*
  - Repeat for all nonzero *k* in **`pnb`**
  - Pass control to next instruction in logical flow for *p*

Default Actions

- **Each signal type has a predefined *default action*, which is one of:**
  - The process terminates
  - The process terminates and dumps core
  - The process stops until restarted by a SIGCONT signal
    For example in shell, bringing a stopped process to foreground or background
  - The process ignores the signal

Installing Signal Handlers

- **The `signal` function modifies the default action associated with the receipt of signal `signum`, and returns the current one**
  - **`handler_t *signal(int signum, handler_t *handler)`**

- **Different values for `handler`:**
  - SIG_IGN: ignore signals of type **`signum`**
  - SIG_DFL: revert to the default action on receipt of signals of type **`signum`**
  - Otherwise, **`handler`** is the address of a ***signal handler***
    - Called when process receives signal of type **`signum`**
    - Referred to as ***"installing"*** the handler
    - Executing handler is called ***"catching"*** or ***"handling"*** the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

A Program That Reacts to Externally Generated Events (Ctrl-c)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
  safe_printf("You think hitting ctrl-c will stop the bomb?\n");
  sleep(2);
  safe_printf("Well...");
  sleep(1);
  printf("OK\n");
  exit(0);
}

main() {
  signal(SIGINT, handler); /* installs ctl-c handler */
  while(1) {
  }
}
```

```
linux> ./external
<ctrl-c>
You think hitting ctrl-c will stop
the bomb?
Well...OK
linux>
```

external.c

A Program That Reacts to Internally Generated Events

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  safe_printf("BEEP\n");

  if (++beeps < 5)
    alarm(1);
  else {
    safe_printf("BOOM!\n");
    exit(0);
  }
}
```

internal.c

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

```
linux> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```

Signal Handling Example

```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    // Concurrency, Reentrant version of printf
    exit(0);
}
void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            while(1); /* child infinite loop
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Signal Handling Example

```
void int_handler(int sig) {
    safe_printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0
            while(1); /* child inf
        }
    for (i = 0; i < N; i++) {
        printf("Killing process %d
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_s
        if (WIFEXITED(child_status
            printf("Child %d termi
                   wpid, WEXITSTAT
        else
            printf("Child %d termi
    }
}
```

```
linux> ./forks 13
Killing process 25417
Killing process 25418
Killing process 25419
Killing process 25420
Killing process 25421
Process 25417 received signal 2
Process 25418 received signal 2
Process 25420 received signal 2
Process 25421 received signal 2
Process 25419 received signal 2
Child 25417 terminated with exit status 0
Child 25418 terminated with exit status 0
Child 25420 terminated with exit status 0
Child 25419 terminated with exit status 0
Child 25421 terminated with exit status 0
linux>
```

Single Signal

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    safe_printf(
           "Received signal %d from process %d\n",
           sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1); /* deschedule child */
            exit(0);  /* Child: Exit */
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}
```

- **Pending signals are not queued**
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal
  - This program may get stuck in final loop

Signal Handler Funkiness

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    safe_printf(
           "Received signal %d from process %d\n",
           sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i+
        if ((pid[i] = fo
            sleep(1); /*
            exit(0);  /*
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}
```

- **Pending signals are not queued**
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal
  - This program may get stuck in final loop

```
linux> ./forks 14
Received SIGCHLD signal 17 for process 21344
Received SIGCHLD signal 17 for process 21345
```

Living With Signle Signals

- **Must wait for all terminated jobs**
  - Have loop with **`waitpid`** to get all jobs
  - -1 for pid: wait for any child process.
  - WNOHANG for option: (wait no hang) return immediately if no child has exited.

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    int n = 0;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        safe_printf("Received signal %d from process %d.  n = %d\n",
                    sig, pid, n++);
    }
}

void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

Living With Single Signals

- **Must wait for all terminated jobs**
  - Have loop with **`waitpid`** to get all jobs

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    int n = 0;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        safe_printf("Received signal %d from process %d.  n = %d\n",
                    sig, pid, n++);
    }
}

void fork15()
{
    . . .
    signal(SIGC
    . . .
}
```

```
greatwhite> forks 15
Received signal 17 from process 27476.  n = 0
Received signal 17 from process 27477.  n = 0
Received signal 17 from process 27478.  n = 0
Received signal 17 from process 27479.  n = 1
Received signal 17 from process 27480.  n = 0
greatwhite>
```

Explicitly Blocking and Unblocking Signals

- The ***sigprocmask*** function changes the set of currently blocked signals. The specific behavior depends on the value of how:
  - SIG_BLOCK: Add the signals in set to blocked (blocked = blocked | set).
  - SIG_UNBLOCK: Remove the signals in set from blocked (blocked = blocked & ~set).
  - SIG_SETMASK: blocked = set.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

Returns: 0 if OK, −1 on error

```
int sigismember(const sigset_t *set, int signum);
```

Returns: 1 if member, 0 if not, −1 on error

Explicitly Blocking and Unblocking Signals

■If oldset is non-NULL, the previous value of the blocked bit vector is stored in oldset.

■The ***sigemptyset*** initializes set to the empty set. The ***sigfillset*** function adds every signal to set. The ***sigaddset*** function adds signum to set, ***sigdelset*** deletes signum from set, and ***sigismember*** returns 1 if signum is a member of set, and 0 if not.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

Returns: 0 if OK, −1 on error

```
int sigismember(const sigset_t *set, int signum);
```

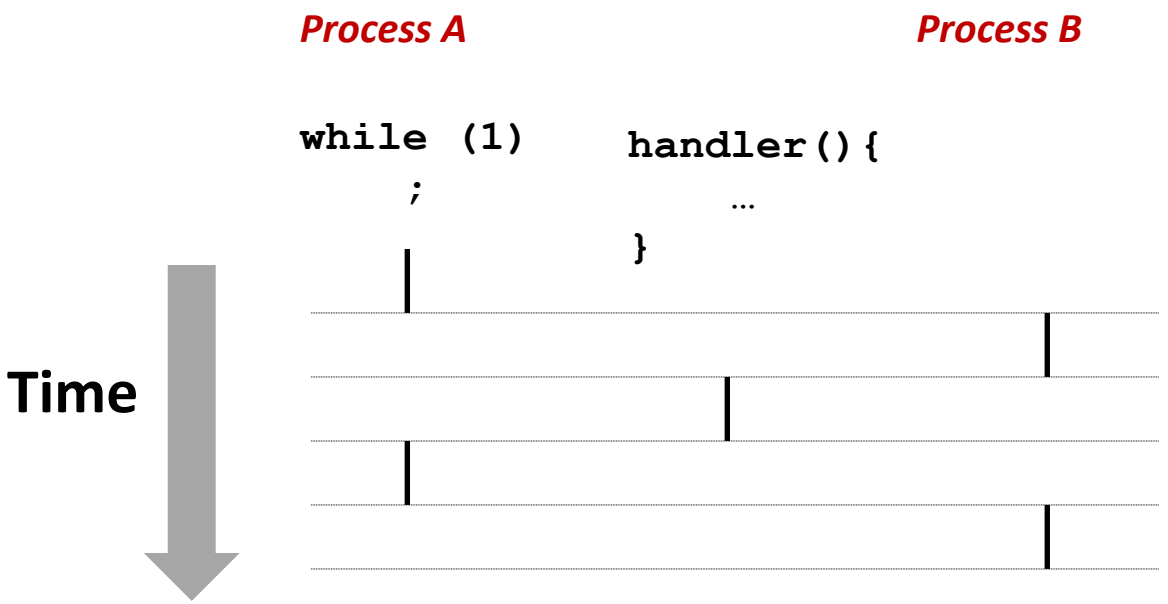
Returns: 1 if member, 0 if not, −1 on error

Race Condition

```
1   void handler(int sig)
2   {
3       pid_t pid;
4       while ((pid = waitpid(-1, NULL, 0)) > 0) /* Reap a zombie chi
5           deletejob(pid); /* Delete the child from the job list */
6
7
8   }
9
10  int main(int argc, char **argv)
11  {
12      int pid;
13
14      Signal(SIGCHLD, handler);
15      initjobs(); /* Initialize the job list */
16
17      while (1) {
18          /* Child process */
19          if ((pid = Fork()) == 0) {
20              Execve("/bin/date", argv, NULL);
21          }
22
23          /* Parent process */
24          addjob(pid);  /* Add the child to the job list */
25      }
26      exit(0);
27  }
```

Signals Handlers as Concurrent Flows

■A signal handler is a separate logical flow (not another process) that runs concurrently with the main program



Eliminate Race Condition by Blocking Signals

```
1   void handler(int sig)
2   {
3       pid_t pid;
4       while ((pid = waitpid(-1, NULL, 0)) > 0) /* Reap a zombie child */
5           deletejob(pid); /* Delete the child from the job list */
6
7
8   }
9
10  int main(int argc, char **argv)
11  {
12      int pid;
13      sigset_t mask;
14
15      Signal(SIGCHLD, handler);
16      initjobs(); /* Initialize the job list */
17
18      while (1) {
19          Sigemptyset(&mask);
20          Sigaddset(&mask, SIGCHLD);
21          Sigprocmask(SIG_BLOCK, &mask, NULL); /* Block SIGCHLD */
22
23          /* Child process */
24          if ((pid = Fork()) == 0) {
25              Sigprocmask(SIG_UNBLOCK, &mask, NULL); /* Unblock SIGCHLD */
26              Execve("/bin/date", argv, NULL);
27          }
28
29          /* Parent process */
30          addjob(pid);  /* Add the child to the job list */
31          Sigprocmask(SIG_UNBLOCK, &mask, NULL);  /* Unblock SIGCHLD */
32      }
33      exit(0);
34  }
```