Operating Systems (234123)
Processes & Signals

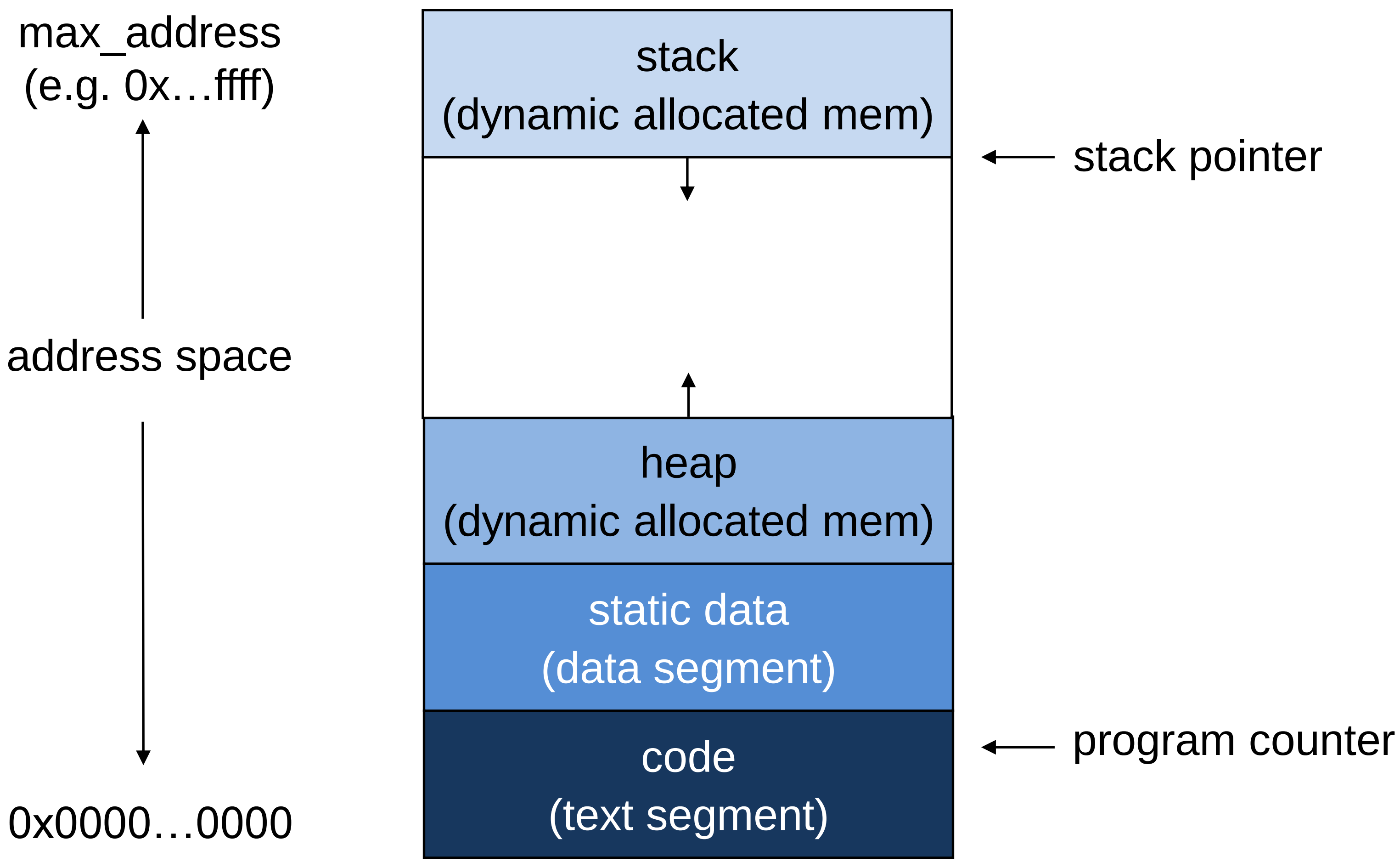
Dan Tsafrir

2024-06-03, 2024-06-10

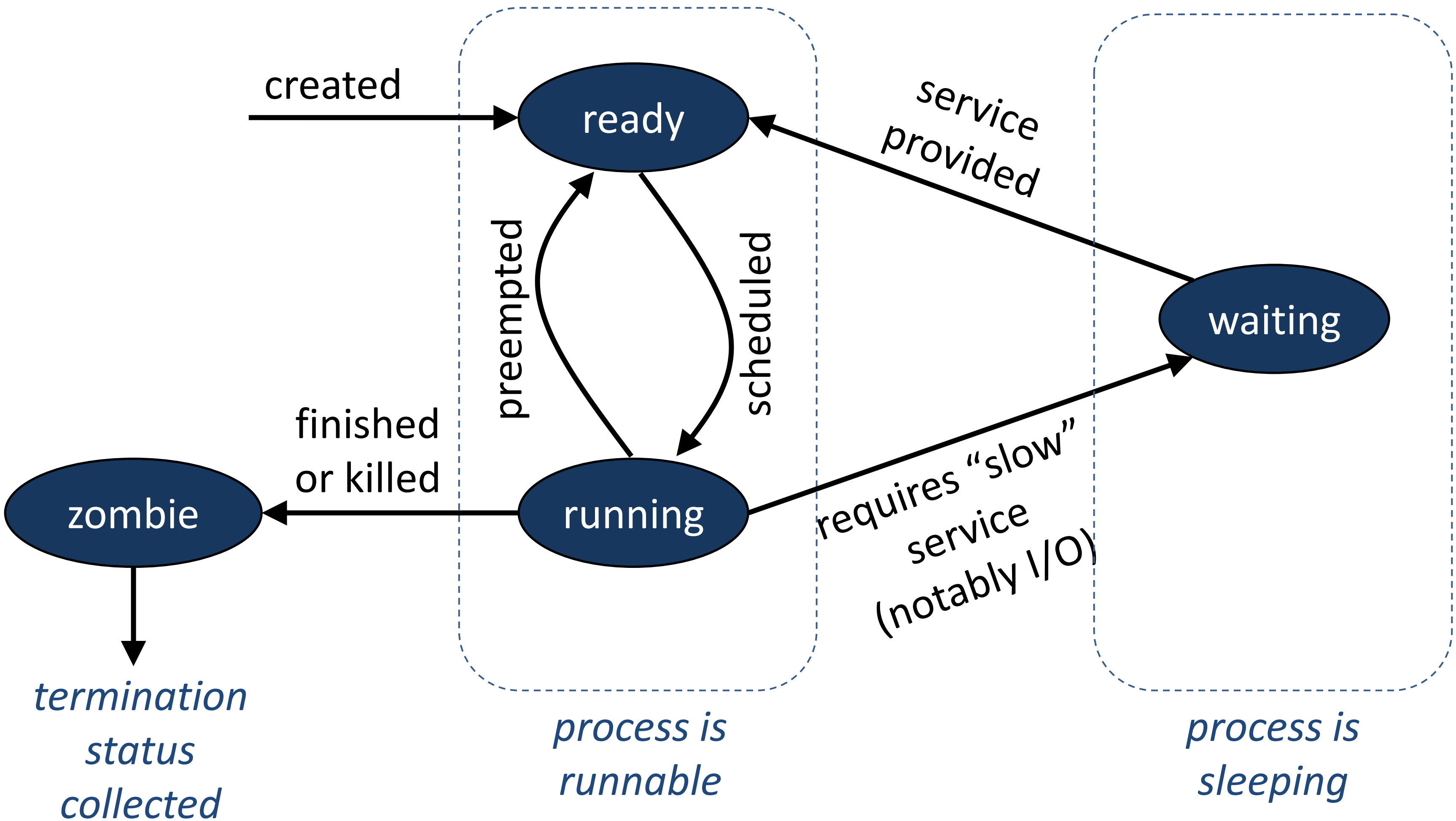
What's a process

- **An implementation of the abstract machine concept**
  - Which we discussed in the previous lecture
- **A running instance of an executable, invoked by a user**
  - Can have multiple independent processes of the same executable
- **A schedulable entity, on the CPU**
  - OS decides which of these entities gets to run on a CPU core, and when
- **Sometimes called**
  - Task or job
- **The OS kernel is neither a process nor a schedulable entity**
  - Rather, it's a set of procedures executing in response to events (≈ interrupts)
  - Albeit sometimes the OS runs some code within schedulable entities
    - But then we prefer not to refer to these entities as "processes", which correspond to *user* programs; we may refer to them as "kernel threads" instead

Process address space is contiguous



Process states



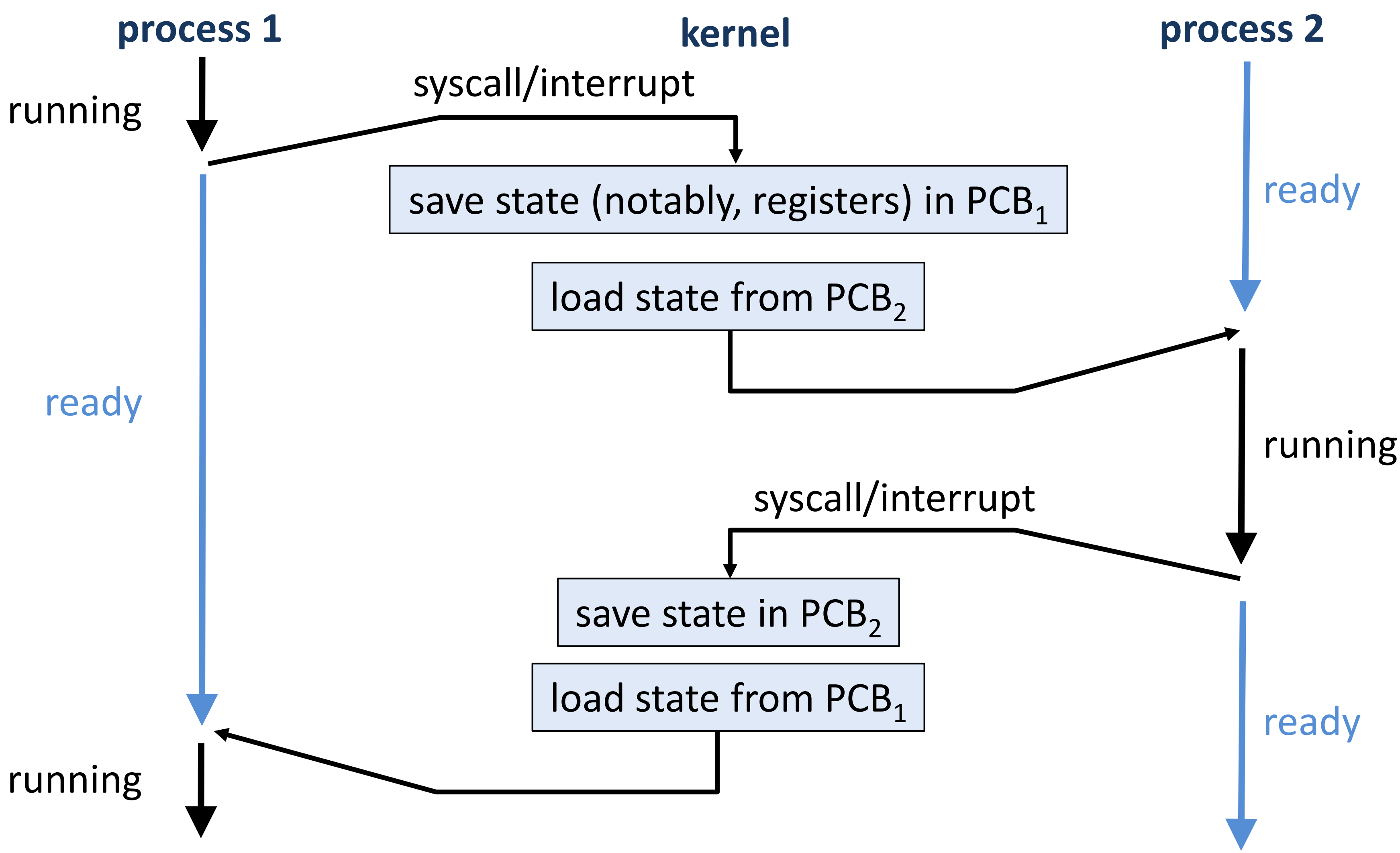
Process control block ("PCB")

- **The OS maintains a "state" for every process**
  - Encapsulated in a PCB
- **In Linux**
  - Called a "process descriptor"
  - Of type task_struct (C struct)
  - Has O(100) fields
- **Used in context switches**
  - Updated upon preemption
  - Loaded upon resumption
- **Question**
  - Can a process access its own PCB?

- **PID (process ID)**
- **UID (user ID)**
- **Pointer to address space**
- **Registers**
- **Scheduling priority**
- **Resources usage limits (e.g., memory, CPU, num of open files)**
- **Resources consumed**
- **State (previous slide)**
- **Current/present working directory (pwd=cwd)**
- **Open files table**
- **…**

process attributes in PCB

Context switching (in runnable state)



Process creation & termination

- **One process (the "parent") can create another (the "child")**
  - A new PCB is allocated and initialized
  - Homework: run 'ps auxwww' in the shell; PPID is the parent's PID
- **In POSIX, child process inherits most of parent's attributes**
  - UID, open files (should be closed if unneeded; why?), cwd, etc.
- **While executing, PCB moves between different queues**
  - According to state change graph
  - Queues: runnable, sleep/wait for event i (i=1,2,3…)
- **After a process dies (exit()s / interrupted), it becomes a zombie**
  - Parent uses wait* syscall to clear zombie from the system (why?)
  - Wait syscall family: wait, waitpid, waitid, wait3, wait4; example:
  - pid_t wait4(pid_t, int *wstatus, int options, struct rusage *rusage);
- **Parent can sleep/wait for its child to finish or run in parallel**
  - wait*() will block unless WNOHANG given in 'options'
  - Homework: read 'man 2 wait'

fork() – spawn a child process

- **fork() initializes a new PCB**
  - Based on parent's value
  - PCB added to runnable queue
- **Now there are 2 processes**
  - At same execution point
- **Child's new address space**
  - Complete copy of parent's space, with one difference…
- **fork() returns twice**
  - At the parent, with pid>0
  - At the child, with pid=0
- **What's the printing order?**
- **'errno' – a global variable**
  - Holds error num of last syscall

```
int main(int argc, char *argv[])
{
  int pid = fork();
  if( pid==0 ) {
      //
      // child
      //
      printf("parent=%d son=%d\n",
              getppid(), getpid());
  }
  else if( pid > 0 ) {
      //
      // parent
      //
      printf("parent=%d son=%d\n",
              getpid(), pid);
  }
  else { // print string associated
         // with errno
      perror("fork() failed");
  }
  return 0;
}
```

System call errors

```
// int errno = number of last system call error.
// Errors aren't zero. (If you want to test value of
// errno after a system call, need to zero it before.)
#include <errno.h> // see man 3 errno

// const char * const sys_errlist[];
// char* strerror(int errnum) {
//     // check errnum is in range
//     return sys_errlist[errnum];
// }
#include <string.h>

//  void perror(const char *prefix);
// prints: "%s: %s\n" , prefix, sys_errlist[errno]
#include <stdio.h>
```

exec*() – replace current process image

- **To start an entirely new program**
  - Use the exec*() syscall family; for example:
    - int execv(const char *progamPath, char *const argv[]);
  - Homework: read 'man execv'
- **Semantics**
  - Stops the execution of the invoking process
  - Loads the executable 'programPath'
  - Starts 'programPath', with 'argv' as its argv
  - Never returns (unless fails)
  - *Replaces* the new process; doesn't create a new process
    - In particular, PID and PPID are the same before/after exec*()

Simplistic UNIX shell loop example

Finished here

```
int main(int argc, char *argv[])
{
  for(;;) {
      int stat;
      char **argv;
      char *c = readNextCom(&argv);
      int pid = fork();

      if( pid < 0 ) {
          perror("fork failed");
      }
      else if( pid==0 ) { // child
          execv(c, argv);
          perror("execv failed");
      }
      else { // parent
          if( wait(&stat) < 0 )
              perror("wait failed");
          else
              chkStatus(pid,stat);
          release(argv);
      }
  }
  return 0;
}
```

```
void chkStatus(int pid, int stat)
{
  if( WIFEXITED(stat) ) {
    printf("%d exit code=%d\n",
           pid, WEXITSTATUS(stat));
  }
  else if( WIFSIGNALED(stat) ) {
    // the topic we're going
    // to learn next
    printf("%d died on signal=%d\n",
           pid, WTERMSIG(stat));
  }
  else if
     // a few more options…
}
```

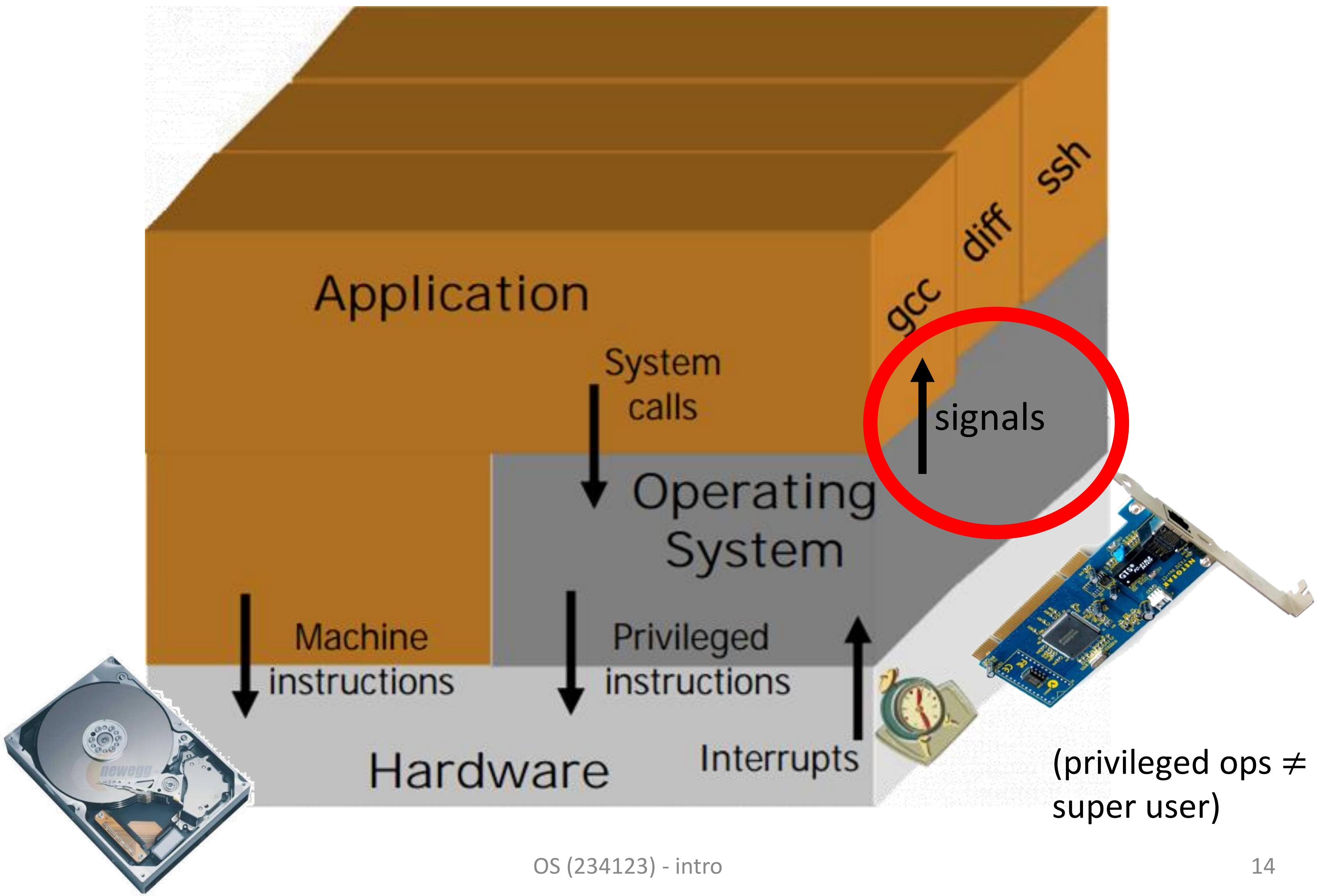
Who wait()-s for an "orphan" process?

- **POSIX specification says:**
  – *"If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process"*
    - https://pubs.opengroup.org/onlinepubs/9699919799/functions/wait.html
- **Linux manual says:**
  – *"init [...becomes] the parent of all processes whose natural parents have died, and it is responsible for reaping those when they die. [...] init expects to have a process id of 1"*
    - https://linux.die.net/man/8/init
  – *"If a parent process terminates, then its zombie children (if any) are adopted by init"*
    - https://linux.die.net/man/2/wait

OS-supported asynchronous notifications

POSIX SIGNALS

Reminder from the 1st lecture



(privileged ops ≠ super user)

What are signals & signal handlers

- **Signal = notification "sent" to a process**
  - To asynchronously notify it that some event occurred
- **When receiving a signal**
  - The process stops whatever it is doing & handles it
- **Default signal handling action**
  - Either die or ignore (depends on the type of the signal)
- **The process can configure how it handles most signals**
  - Different signals can have a different handlers,
    and they can be temporarily blocked/unblocked = "**masked"/"unmasked**"
    - Except for 2 signals (well, actually 3 – discussed shortly)
- **Signals have names and numbers standardized by POSIX**
  - Do 'man 7 signal' in shell/Google for a listing/explanation of all signals
  - For example: http://man7.org/linux/man-pages/man7/signal.7.html ,
    https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html
  - **HOMEWORK**: take a few minutes to quickly survey all signals

Silly example

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

void sigfpe_handler(int signum) {
    fprintf(stderr,"I divided by zero (sig=%d)!\n",
            signum);    // prints SIGFPE's const value
    exit(EXIT_FAILURE); // what happens if not exiting?
}

int main() {
    signal(SIGFPE, sigfpe_handler);
    int x = 1/0; // processor interrupt, then OS signal
    return 0;
}
```

Another silly example

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

void sigint_handler(int signum) {
    printf("I’m disregarding your crtl-c!\n");
}

int main() {
    // when pressing ctrl-c in the shell
    // => SIGINT is delivered to foreground process
    // (who makes this happen?)
    signal(SIGINT,sigint_handler);
    for(;;) { /*endless loop*/ }
    return 0;
}
```

Another silly example

```
<0>dan@csa:~$ ./a.out
# here I clicked ctrl-c => delivered SIGINT
I'm disregarding your crtl-c!
# here I clicked ctrl-c again => delivered SIGINT
I'm disregarding your crtl-c!
# here I clicked ctrl-z => delivered SIGSTOP, must obey
[1]+  Stopped                   ./a.out
<148>dan@csa:~$ ps
  PID TTY          TIME CMD
10148 pts/19   00:00:00 bash
21709 pts/19   00:00:12 a.out
21710 pts/19   00:00:00 ps
<0>dan@csa:~$ kill -9 21709    # 9=SIGKILL, must obey
[1]+  Killed                    ./a.out
<0>dan@csa:~$
```

Another silly example

```
<0>dan@csa:~$ ./a.out
# here I clicked ctrl-c (=> deliver SIGINT)
I'm ignoring your crtl-c!
# here I                                       NT)
I'm ignor
# here I                                       't ignore
[1]+  Sto
<148>dan@
  PID TTY          TIME CMD
10148 pts/19   00:00:00 bash
21709 pts/19   00:00:12 a.out
21710 pts/19   00:00:00 ps
<0>dan@csa:~$ kill -9 21709    # 9=SIGKILL, can't ignore
[1]+  Killed                   ./a.out
<0>dan@csa:~$
```

When I click ctrl-c, the OS gets an interrupt from the keyboard, which the OS then translates into a SIGINT signal delivered to the relevant process.

Notice

- **Argument for the 'kill' shell utility**
  - Any signal (not just 9=kill)
    - kill -9 <pid>
    - kill -s KILL <pid>
    - kill -s SIGKILL <pid>
- **There are 2+1=3 signals that a process can't ignore**
  - SIGKILL = terminate the receiving process
  - SIGSTOP = suspend the receiving process (make it sleep)
- **Is the affect of SIGSTOP reversible?**
  - Yes, when you send to the process SIGCONT
- **SIGCONT can't be ignored either…**
  - A SIGSTOP-ed process *will* continue
  - But process can set a handler for it, which will be invoked immediately when the process gets hit by the SIGCONT and is resumed as a result
- **What can you do with SIGSTOP/SIGCONT?**

Job control

- **In the shell, assume we run a program ‘loop’ that does this**
  - int main() { while(1); return 0; }
- **As noted, clicking ctrl-z in the shell**
  - Will make ‘loop’ sleep
- **Subsequently, invoking ‘fg’ in the shell**
  - Will wake ‘loop’ up in the foreground (meaning, the shell sleeps, and any typed input is directed to ‘loop’)
- **Alternatively, invoking ‘bg’ in the shell**
  - Will wake ‘loop’ up in the background (as is we executed it with “&”, so shell becomes operational, and typed input goes to the shell)
- **Simplifying lie I told**
  - For reasons related to job control, actually, ctrl-z
    - Generates STGTSTP, not SIGSTOP
  - There are subtle differences between the two, which we won’t learn

This is how a signal is truly ignored

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

int main() {
    // Before, we used sigint_handler as the arg.
    // Now, we use the SIG_IGN macro, which means
    // no handler will be called.
    // There's also SIG_DFL, to restore the default
    // behavior
    signal(SIGINT, SIG_IGN);
    for(;;) { /*endless loop*/ }
    return 0;
}
```

Example – ask a running daemon how much "work" it did thus far

- A "daemon" is
  - Background process, not controlled interactively by user
  - In shell: **nohup** <command> & (see: https://linux.die.net/man/1/nohup)
  - Daemon name typically ends with "d" (e.g., sshd, syslogd, swapd)

```
void do_work() { for(int i=0; i<10000000; i++); }
int g_count=0; // counts num. of times do_work was invoked
void sigusr_handler(int signum) {
    printf("Work done so far: %d\n", g_count);
}
int main() {
    signal(SIGUSR1,sigusr_handler);
    for(;;) { do_work(); g_count++; }
    return 0;
}
```

Example – ask a running daemon how much "work" it did thus far

```
<0>dan@csa:~$ ./a.out &
[1] 23998
# recall: kill utility also accepts strings as signals
<0>dan@csa:~$ kill -s USR1 23998
Work done so far: 626
<0>dan@csa:~$ kill -s USR1 23998
Work done so far: 862
<0>dan@csa:~$ kill -s USR1 23998
Work done so far: 1050
<0>dan@csa:~$ kill -s KILL 23998
[1]+  Killed                  ./a.out
```

Enumerate signals

- **SIGSEGV, SIGBUSS, SIGILL, SIGFPE**
  - ILL = illegal instruction (trying to invoke privileged instruction)
  - SEGV = segmentation violation (illegal memory ref, e.g., outside an array)
  - BUS = dereference invalid address (null/misaligned, assume it's like SEGV)
  - FPE = floating point exception (despite name, actually *all* arithmetic errors, nor just floating point; example: divide by zero)
  - These are driven by the associated (HW) interrupts
    - The OS gets the associated interrupt
    - The OS interrupt handler sees to it that the misbehaving process gets the associated signal
    - The default signal handler for these signals: core dump + die
- **SIGCHLD**
  - Parent gets it whenever fork()ed child terminates or is SIGSTOP-ed
- **SIGALRM**
  - Get a signal after some specified time
  - Set by system calls: alarm(2) & setitimer(2) (homework: read man)

Enumerate signals

- **SIGTRAP**
  - When debugging / single-stepping a process
  - E.g., can be delivered upon each instruction
- **SIGUSR1, SIGUSR2**
  - User decides the meaning (e.g., see our daemon example)
- **SIGXCPU**
  - Delivered when a process used up more CPU then its soft-limit allows
  - Soft/hard limits are set by the system call: setrlimit()
  - Soft-limits warn the process its about to exceed the hard-limit
  - Exceeding the hard-limit => SIGKILL will be delivered
- **SIGPIPE**
  - Write to pipe with no readers (we'll learn about pipes later, for the time being think about the shell's pipe: "|")

Enumerate signals

- **SIGIO**
  - Can configure file descriptors such that a signal will be delivered whenever some I/O is ready
  - Typically makes sense when also configuring the file descriptors to be "non blocking"
    - E.g., when read()ing from a non-blocking file descriptor, the system call immediately returns to user if there's currently nothing to read
    - In this case, errno will be set to EAGAIN = EWOULDBLOCK
- **And a few more**
  - man 7 signal
  - http://man7.org/linux/man-pages/man7/signal.7.html

Signal vs. interrupts

	interrupts	signals
Who triggers them? Who defines their meaning?	Hardware: CPU cores (sync) & other devices (async)	Software (OS), HW is unaware
Who handles them? Who (un)blocks them?	OS	processes
When do they occur?	Both synchronously & asynchronously	Likewise, but, technically, invoked when returning from kernel to user

Signal system calls – sending

- **int kill(pid_t *pid*, int *sig*)**
  - (Not the shell utility, the actual system call)
  - Allows a process to send a signal to another process (or to itself)
    - Homework: How?
  - man 2 kill – http://linux.die.net/man/2/kill
    - “2” is for system calls
    - “1” is for shell utilities

Signal system calls – (un)blocking

- **Signals can be asynchronous => might lead to ”race conditions”**
  - Therefore, as noted, all signals (except kill/stop) can be blocked
  - Like how OS disables/enables interrupt
- **How**
  - The PCB maintains a set of currently blocked signals
  - Which can be manipulated by users via the following syscall
- **int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oldset*)**
  - ‘how’ = SIG_BLOCK (+=), SIG_UNBLOCK (-=), SIG_SETMASK (=)
  - ‘set’ can be maipulated with sigset ops sig**etmpty**set(sigset_t *set), sig**fill**set(sigset_t *), sig**add**set(sigset_t *set, int signum), sig**is**member(sigset_t *set, int signum)
- **Manual**
  - man 2 sigprocmask – http://linux.die.net/man/2/sigprocmask
  - man 3 sigsetops – https://linux.die.net/man/3/sigsetops

(Un)blocking example

```
Record_t db[N];

void my_handler(int signum) {
  /* may read/update db */
}

int main(int argc, char *argv[])
{
  sigset_t mask, orig_mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGTERM);

  signal(SIGTERM, my_handler);

  for(;;) {
    char *cmd = read_command();
    sigprocmask(SIG_BLOCK, &mask, &orig_mask);
    // do stuff that may read/update db…
    sigprocmask(SIG_SETMASK, &orig_mask, NULL);
  }
  return 0;
}
```

Recall that every syscall might fail; the example ignores this for brevity

Signal system calls – control & more info

- **Additional info about signal & fine-grain control**
  - over how signals operate is provided via the following syscall
- **int sigaction(int signum,**
  const struct sigaction *act,
  struct sigaction *oldact)
  - man 2 sigaction – http://linux.die.net/man/2/sigaction
    (homework: **read it**)

Signals interact with other system calls

- **ssize_t read(int fd, void *buf, size_t count);**
  - What happens if getting signal while read()ing?
  - The read system call returns -1, and it sets the global variable 'errno' to hold EINTR
  - An example whereby read() might fail and user should simply retry

```
int readn( int sockfd /*learn later*/, char *ptr, int nbytes )
{
  int nleft = nbytes;

  while( nleft > 0 ) {  // 'read' is typically done in a loop (why?)
        int nread = read(sockfd, ptr, nleft);
        if( (nread == -1) && (errno != EINTER) ) {
            fprintf(stderr, "read failed, errno=%m", errno);
            return -1;
        }
        else if( nread == 0 )
           break; /*EOF*/

        nleft -= nread;
        ptr   += nread;
  }
  return nbytes - nleft;
}
```